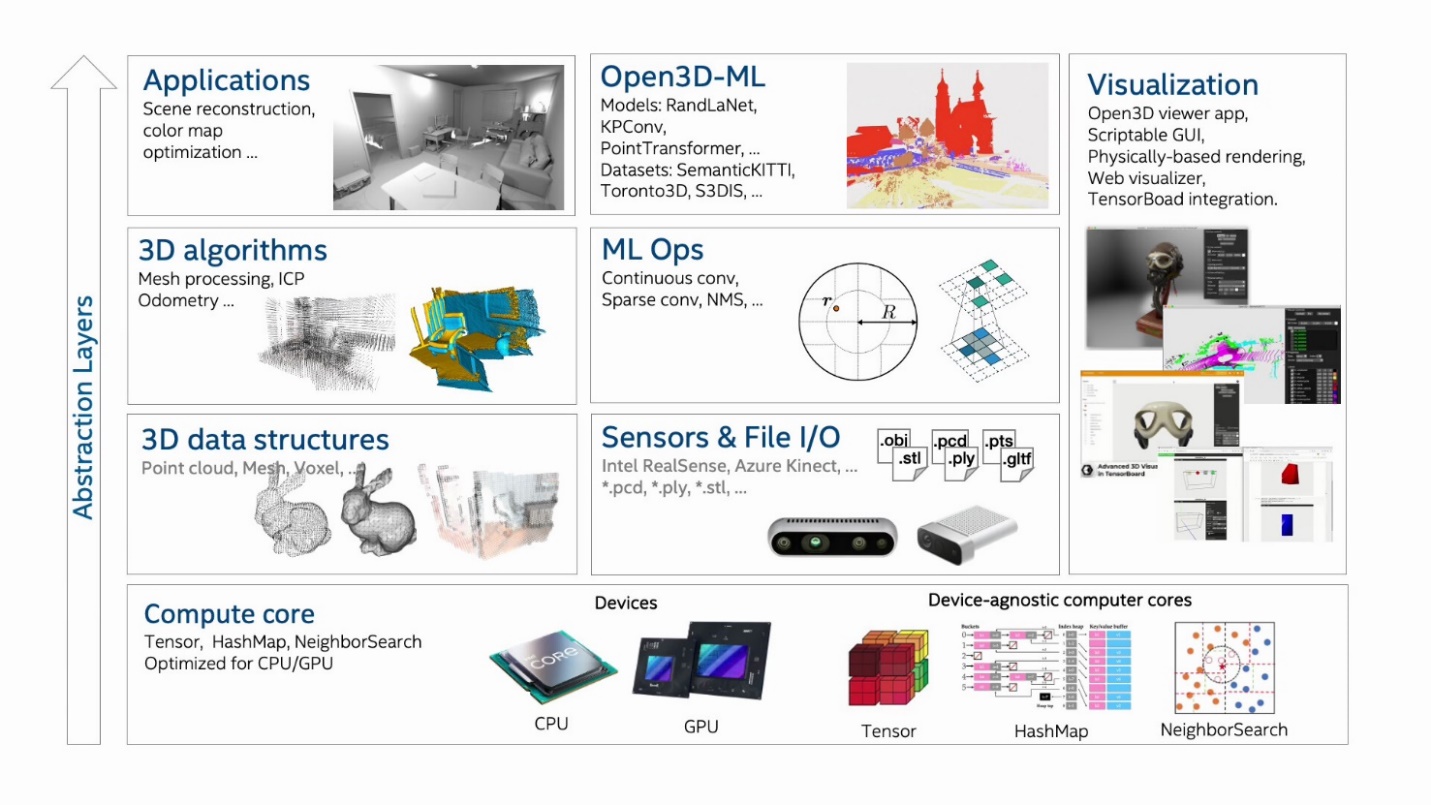
Open3D Documentation Summary

<https://www.open3d.org/docs/release/>

# Getting Started

## Introduction

Here’s a brief overview of the different components of **Open3D** and how they fit together to enable full end to end pipelines:



**Open3D-ML** is an extension of Open3D for 3D machine learning tasks. It builds on top of the Open3D core library and extends it with machine learning tools for 3D data processing.

Check it out at [Open3D-ML](https://github.com/isl-org/Open3D-ML).

## Q: Claude: What is Open3D?

Open3D is a powerful, open-source library designed specifically for working with 3D data in computer vision, robotics, and graphics applications. Think of it as a comprehensive toolkit that gives developers and researchers everything they need to handle 3D information - from simple point clouds to complex geometric meshes.

Let me help you understand this by building up from the basics. In our everyday world, we naturally perceive depth and three-dimensional structure, but computers need special tools to process this kind of spatial information. Open3D bridges this gap by providing a unified framework for 3D data processing.

The library excels in several key areas. First, it handles point clouds - imagine millions of tiny dots in 3D space that together represent the surface of an object, like what you might get from a LiDAR scanner or depth camera. Open3D can read, write, visualize, and process these point clouds with operations like filtering noise, finding patterns, or reconstructing surfaces.

Second, Open3D works extensively with 3D meshes - these are collections of connected triangles that form the surface of 3D objects, similar to how a wireframe model represents a car or building. The library provides tools for mesh generation, simplification, and analysis.

What makes Open3D particularly valuable is its dual nature. It offers both Python and C++ interfaces, making it accessible to researchers who prefer Python's simplicity while still providing the performance benefits of C++ for computationally intensive tasks. This design philosophy means you can prototype quickly in Python and then optimize performance-critical sections in C++ if needed.

The library shines in practical applications like 3D reconstruction (building 3D models from multiple camera views), registration (aligning different 3D scans of the same object), and visualization (creating interactive 3D displays). For instance, archaeologists might use Open3D to reconstruct ancient artifacts from photogrammetry data, while autonomous vehicle developers might use it to process LiDAR sensor data for navigation.

Open3D also integrates well with the broader scientific Python ecosystem, working alongside libraries like NumPy for numerical computing and matplotlib for 2D plotting, creating a seamless workflow for 3D data analysis.

### Usage for robot hand

Q: Can you tell me more about how to use Open3D in robotics to process data from 3D cameras. Eg to handle some object by a robotic hand (for which we'll need to calculate the trajectory)

Using Open3D for robotic manipulation is a fascinating application that combines computer vision, spatial understanding, and motion planning into one cohesive system. Let me walk you through how this works, building from the fundamental concepts to the complete pipeline you'd need for a robotic hand to grasp and manipulate objects.

Think of this process like teaching a robot to see and understand the world the way you do when you reach for a coffee cup. Your brain instantly processes the cup's shape, orientation, surface properties, and your hand's relationship to it. We need to give the robot similar capabilities through systematic data processing.

Understanding the Input: What 3D Cameras Provide

First, let's establish what we're working with. 3D cameras like Intel RealSense, Microsoft Kinect, or stereo camera systems provide two main types of data. The depth information tells us how far each pixel is from the camera, essentially creating a 3D point cloud where each point has coordinates in real-world space. The RGB information gives us color and texture details that help with object recognition and surface understanding.

When Open3D receives this data, it typically comes as organized point clouds - imagine a structured grid of 3D points that maintains the spatial relationships from the original camera view. This organization is crucial because it preserves neighborhood information that we'll need for processing.

The Processing Pipeline: From Raw Data to Understanding

The first step in any robotic manipulation pipeline involves cleaning and preparing the data. Raw 3D camera data often contains noise, outliers from reflective surfaces, and gaps where the sensor couldn't measure depth. Open3D provides several filtering techniques here. Statistical outlier removal identifies points that are unusually far from their neighbors - these often represent sensor errors. Radius outlier removal eliminates isolated points that don't have enough neighbors within a reasonable distance.

Here's where Open3D's strength becomes apparent. The library allows you to chain these operations efficiently while maintaining the point cloud's spatial integrity. For instance, you might first apply a bilateral filter to smooth the data while preserving sharp edges that represent object boundaries, then remove statistical outliers to clean up sensor noise.

Segmentation: Identifying Objects in the Scene

Once you have clean data, the next challenge is segmentation - separating the object you want to manipulate from everything else in the scene. This is where the robot begins to "understand" what it's looking at, similar to how your visual system automatically separates the coffee cup from the table it sits on.

Open3D provides several segmentation approaches. Plane segmentation using RANSAC (Random Sample Consensus) is particularly useful for removing background surfaces like tables or floors. The algorithm repeatedly samples small sets of points, fits planes to them, and identifies which plane has the most supporting points. This effectively separates horizontal surfaces from the objects resting on them.

For more complex scenes, clustering algorithms like DBSCAN group nearby points together, helping identify individual objects. The beauty of Open3D's implementation is that these algorithms work directly with 3D spatial relationships, not just 2D image patterns, giving you much more robust object separation.

Surface Reconstruction and Mesh Generation

Raw point clouds, while useful, don't directly tell you about object surfaces and their properties. This is where surface reconstruction becomes critical for manipulation planning. Think of it as the difference between knowing where the dots are on a connect-the-dots puzzle versus understanding the complete picture they form.

Open3D's Poisson surface reconstruction is particularly elegant here. It treats the surface reconstruction problem as solving a spatial function where the object's surface is the boundary between inside and outside regions. This gives you smooth, watertight meshes that accurately represent object surfaces, which is essential for calculating contact points and grip forces.

Normal Estimation and Surface Properties

For a robotic hand to grasp an object effectively, it needs to understand surface orientations and properties. Surface normals - vectors perpendicular to the surface at each point - tell the robot which direction is "outward" from the object. This information is crucial for determining how to approach the object and where to place fingertips.

Open3D computes these normals by analyzing the local neighborhood around each point, essentially fitting small planes to nearby points and calculating the perpendicular direction. The library also provides tools for orienting these normals consistently, ensuring they all point outward or inward relative to the object.

Grasp Planning Integration

Here's where the spatial understanding provided by Open3D translates into actionable robotic control. The processed 3D data provides several key inputs for grasp planning algorithms. The object's geometric center of mass helps determine stable grip points. Surface curvature information identifies areas suitable for different types of grasps - flat surfaces for pinch grips, curved surfaces for wrap-around grasps.

Open3D's collision detection capabilities become crucial when planning the approach trajectory. The library can efficiently test whether the robot's hand or arm would collide with the object or surrounding environment during the planned motion. This happens through fast geometric queries between the robot's mesh model and the reconstructed scene.

Trajectory Calculation and Execution

The final step involves translating this geometric understanding into robot motion. Open3D provides the spatial constraints and target poses, but trajectory planning typically involves additional considerations like joint limits, smooth motion profiles, and dynamic constraints.

The 3D scene understanding from Open3D feeds into motion planning algorithms by providing obstacle representations, target object poses, and approach directions. For instance, if Open3D determines that an object has a cylindrical shape with its axis oriented vertically, the trajectory planner can optimize for a top-down approach that aligns the gripper with this axis.

Real-World Considerations and Challenges

In practice, this pipeline faces several interesting challenges that Open3D helps address. Partial occlusion means you often can't see the complete object - the robot might need to plan grasps based on incomplete information. Open3D's surface reconstruction capabilities can help interpolate missing regions based on visible surface patterns.

Dynamic scenes where objects move or change require real-time processing capabilities. Open3D's efficient algorithms and GPU acceleration options make it possible to update object understanding at rates suitable for reactive robotic control.

Integration with Robotic Frameworks

Open3D integrates well with robotic development frameworks like ROS (Robot Operating System), allowing the 3D processing pipeline to communicate seamlessly with robot control systems. The library's data structures convert readily to and from standard robotic message formats, enabling smooth integration with existing robotic software stacks.

The key insight is that Open3D transforms the raw sensory chaos of 3D camera data into structured geometric understanding that robotic systems can reason about and act upon. It bridges the gap between sensing and manipulation, providing the spatial intelligence that enables robots to interact meaningfully with their three-dimensional world.

Would you like me to dive deeper into any particular aspect of this pipeline? For instance, we could explore specific algorithms for grasp point detection, or examine how real-time constraints affect the choice of processing techniques in different robotic applications.

# Tutorial

## Core

### Tensor

<https://www.open3d.org/docs/release/tutorial/core/tensor.html>

#### Tensor according to ChatGPT :

In the context of 3D graphics development, a tensor is generally understood as a mathematical object used to represent data with multiple dimensions.

**1.** Mathematical Background

A tensor is a generalization of scalars (0D), vectors (1D), and matrices (2D) to N-dimensional arrays of numbers.

**2.** Common Uses in 3D Graphics

While "tensor" might not always be explicitly mentioned, the concepts are heavily used:

* **Transformations**: Matrices (2D tensors) are used for affine transformations—translation, rotation, scaling.
* **Mesh Data**: Arrays of vertex attributes (position, normals, UVs) can be viewed as tensors.
* **Textures and Volumetric Data**: 2D and 3D textures are stored as tensors—e.g., a 3D texture is a 3D tensor of color values.
* **Physics Simulations**: Tensor math is used to simulate forces, deformation, and momentum.
* **Machine Learning/Neural Rendering**: In modern techniques like Neural Radiance Fields (NeRFs), tensors represent scene data, camera parameters, and more.

#### Tensor creation

Tensor can be created from list, numpy array, another tensor.

# Tensor from list.

a = o3c.Tensor([0, 1, 2])

print("Created from list:\n{}".format(a))

# Tensor from Numpy.

a = o3c.Tensor(np.array([0, 1, 2]))

print("\nCreated from numpy array:\n{}".format(a))

# Dtype and inferred from list.

a\_float = o3c.Tensor([0.0, 1.0, 2.0])

print("\nDefault dtype and device:\n{}".format(a\_float))

# Specify dtype.

a = o3c.Tensor(np.array([0, 1, 2]), dtype=o3c.Dtype.Float64)

print("\nSpecified data type:\n{}".format(a))

# Specify device.

a = o3c.Tensor(np.array([0, 1, 2]), device=o3c.Device("CUDA:0"))

print("\nSpecified device:\n{}".format(a))

#### What "Device" Means according to Claude

In computational frameworks like Open3D, PyTorch, or TensorFlow, a "device" refers to the hardware component that will perform the actual computations:

o3c.Device("CUDA:0") specifies:

* **CUDA**: Use NVIDIA GPU with CUDA support
* **:0**: Use the first GPU (if you have multiple GPUs, you could specify "CUDA:1", "CUDA:2", etc.)

Tensor can also be created from another tensor by invoking the copy constructor. This is a shallow copy, the data\_ptr will be copied but the memory it points to will not be copied.

# Shallow copy constructor.

vals = np.array([1, 2, 3])

src = o3c.Tensor(vals)

dst = src

#### Copy & Device transfer

# Host -> Device.

a\_cpu = o3c.Tensor([0, 1, 2])

a\_gpu = a\_cpu.cuda(0)

print(a\_gpu)

# Device -> Host.

a\_gpu = o3c.Tensor([0, 1, 2], device=o3c.Device("CUDA:0"))

a\_cpu = a\_gpu.cpu()

print(a\_cpu)

# Device -> copy in same or another Device.

a\_gpu\_0 = o3c.Tensor([0, 1, 2], device=o3c.Device("CUDA:0"))

a\_gpu\_1 = a\_gpu\_0.cuda(0)

print(a\_gpu\_1)

#### Type casting

We can cast tensor’s data type. Forced casting might result in data loss.

# E.g. float -> int

a = o3c.Tensor([0.1, 1.5, 2.7])

b = a.to(o3c.Dtype.Int32)

#### Numpy I/O with direct memory map

Tensors created by passing numpy array to the constructor (o3c.Tensor(np.array(...)) do not share memory with the numpy array. To have shared memory, you can use o3c.Tensor.from\_numpy(...) and o3c.Tensor.numpy(...).

#### PyTorch I/O with DLPack memory map

We can convert tensors from/to DLManagedTensor.

Q Claude: What is DLManagedTensor and DLPack ?

DLManagedTensor and DLPack are key components of an open standard for sharing tensor data between different deep learning frameworks without copying the underlying data.

**DLPack** is a specification that defines a common in-memory tensor structure. It was created to solve the problem of efficiently sharing tensors between different frameworks like PyTorch, TensorFlow, Apache MXNet, and others. Before DLPack, moving data between frameworks required expensive memory copies and format conversions.

**DLManagedTensor** is the core data structure in DLPack. It's a C struct that contains:

* **DLTensor**: The actual tensor metadata including data pointer, shape, strides, data type, and device information
* **Manager context**: A void pointer to framework-specific context
* **Deleter function**: A function pointer that knows how to properly clean up the tensor when it's no longer needed

The "managed" aspect is crucial because it handles memory management across framework boundaries. When a tensor is shared from one framework to another, the receiving framework gets a DLManagedTensor that includes instructions on how to properly deallocate the memory when done.

#### Reduction:

Open3D supports following reduction operations.

1. **sum** - returns a tensor with sum of values over a given axis.
2. **mean** - returns a tensor with mean of values over a given axis.
3. **prod** - returns a tensor with product of values over a given axis.
4. **min** - returns a tensor of minimum values along a given axis.
5. **max** - returns a tensor of maximum values along a given axis.
6. **argmin** - returns a tensor of minimum value indices over a given axis.
7. **argmax** - returns a tensor of maximum value indices over a given axis.

Examples

vals = np.array(range(24)).reshape((2, 3, 4))

a = o3c.Tensor(vals)

print("a.sum = *{}***\n**".format(a.sum()))

print("a.min = *{}***\n**".format(a.min()))

print("a.ArgMax = *{}***\n**".format(a.argmax()))

# With specified dimension.

vals = np.array(range(24)).reshape((2, 3, 4))

a = o3c.Tensor(vals)

print("Along dim=0**\n***{}*".format(a.sum(dim=(0))))

print("Along dim=(0, 2)**\n***{}***\n**".format(a.sum(dim=(0, 2))))

#### Slicing, indexing, getitem, and setitem

Basic slicing is done by passing an integer, slice object (start:stop:step), index array or boolean array. Slicing and indexing produce a view of the tensor. Hence any change in it will also get reflected in the original tensor.

#### Advanced indexing

Advanced indexing is triggered while passing an index array or a boolean array or their combination with integer/slice object. Note that advanced indexing always returns a copy of the data (contrast with basic slicing that returns a view).

#### Logical operations

Open3D supports following logical operators:

1. **logical\_and** - returns tensor with element wise logical AND.
2. **logical\_or** - returns tensor with element wise logical OR.
3. **logical\_xor** - returns tensor with element wise logical XOR.
4. **logical\_not** - returns tensor with element wise logical NOT.
5. **all** - returns true if all elements in the tensor are true.
6. **any** - returns true if any element in the tensor is true.
7. **allclose** - returns true if two tensors are element wise equal within a tolerance.
8. **isclose** - returns tensor with element wise allclose operation.
9. **issame** - returns true if and only if two tensors are same(even same underlying memory).

#### Comparison Operations

a = o3c.Tensor([0, 1, -1])

b = o3c.Tensor([0, 0, 0])

print("a > b = *{}*".format(a > b))

print("a >= b = *{}*".format(a >= b))

print("a < b = *{}*".format(a < b))

print("a <= b = *{}*".format(a <= b))

print("a == b = *{}*".format(a == b))

print("a != b = *{}*".format(a != b))

#### Nonzero operations

When as\_tuple is False (default), it returns a tensor indices of the elements that are non-zero. Each row in the result contains the indices of a non-zero element in the input.

a = o3c.Tensor([[3, 0, 0], [0, 4, 0], [5, 6, 0]])

print("a.nonzero() = **\n***{}***\n**".format(a.nonzero()))

#### Pickle support

Since Open3D v0.16.0, tensor can be serialized and deserialized using pickle. This is useful for saving and loading tensors to/from disk.

b = pickle.load(open(file\_name, 'rb'))

### Hash Map

<https://www.open3d.org/docs/release/tutorial/core/hashmap.html>

A hash map is a data structure that maps keys to values with amortized O(1) insertion, find, and deletion time. The map is unordered.

Open3D allows parallel hashing on CPU and GPU with keys and values organized as Tensors, where we take a batch of keys and/or values as input.

* **Keys**: The Open3D hash map supports multi-dimensional keys. Due to precision issue, floating-point is not recommended to be regarded as keys.
* **Values**: The Open3D hash map supports values in arbitrary dimensions and data types.
* **Devices**: Both CPU and CUDA are supported. The CPU hashmap is based on [TBB](https://github.com/oneapi-src/oneTBB), while the CUDA hash map is based upon [stdgpu](https://github.com/stotko/stdgpu).

#### According to Claude:

while the fundamental concept of a hash map remains the same, Open3D's implementation has several important differences

The Core Difference: Tensor-Based Architecture

Open3D allows parallel hashing on CPU and GPU with keys and values organized as Tensors, where we take a batch of keys and/or values as input. This is fundamentally different from traditional hash maps that typically work with individual key-value pairs.

Think of it this way: instead of inserting one item at a time into your filing cabinet, imagine you could process an entire box of documents simultaneously. That's what Open3D's tensor-based approach enables.

Multi-Dimensional Keys for 3D Applications

The Open3D hash map supports multi-dimensional keys.

#### A simple example

We first create a simple hash map from integers to integers.

hashmap = o3c.HashMap(capacity,

key\_dtype=o3c.int64,

key\_element\_shape=(1,),

value\_dtype=o3c.int64,

value\_element\_shape=(1,),

device=device)

##### Insertion

Next we show how to insert a batch of (key, value) pairs.

# Prepare a batch of 7 key/values, each a int64 element

keys = o3c.Tensor([[100], [200], [400], [800], [300], [200], [100]],

dtype=o3c.int64,

device=device)

vals = o3c.Tensor([[1], [2], [4], [8], [3], [2], [1]],

dtype=o3c.int64,

device=device)

buf\_indices, masks = hashmap.insert(keys, vals)

##### Query

The query operation follows the similar convention.

query\_keys = o3c.Tensor([[1000], [100], [300], [200], [100], [0]],

dtype=o3c.int64,

device=device)

buf\_indices, masks = hashmap.find(query\_keys)

valid\_keys = query\_keys[masks]

buf\_indices = buf\_indices[masks].to(o3c.int64)

valid\_vals = hashmap.value\_tensor()[buf\_indices]

print('found valid keys: **\n**', valid\_keys)

print('found valid values: **\n**', valid\_vals)

##### Active entries in the hash map

Sometimes we are interested in all the active entries.

**def** print\_active\_entries(hashmap):

active\_buf\_indices = hashmap.active\_buf\_indices().to(o3c.int64)

active\_keys = hashmap.key\_tensor()[active\_buf\_indices]

print('all active keys:**\n**', active\_keys)

active\_vals = hashmap.value\_tensor()[active\_buf\_indices]

print('all active values:**\n**', active\_vals)

##### Erase

We can similarly erase keys. The behavior is similar to insert:

erase\_keys = o3c.Tensor([[100], [1000], [100]], dtype=o3c.int64, device=device)

masks = hashmap.erase(erase\_keys)

##### Activate

In some cases, we know a key is occupied, but do not know the associated value - we prefer to compute and modify it in-place afterwards. This can be achieved by a chain of operations:

activate\_keys = o3c.Tensor([[1000], [0]], dtype=o3c.int64, device=device)

buf\_indices, masks = hashmap.activate(activate\_keys)

buf\_vals = hashmap.value\_tensor()

# Note the assigned tensor has to be strictly in the shape of (N, 1) due to broadcasting

buf\_vals[buf\_indices[masks].to(o3c.int64)] \

= o3c.Tensor([[10], [0]],

dtype=o3c.int64,

device=device)

print\_active\_entries(hashmap)

##### Rehashing and reserve

Rehashing will be automatically triggered when the initial capacity is exceeded after multiple insertions, where the capacity of the hash map is doubled. Rehashing will change the location (i.e. buffer indices) of the inserted key-value pairs, so an update of the buffer indices in the downstream applications is required.

#### Multi-valued hash map

In real-world applications, we want to map coordinates to complex data structures, e.g. a voxel position to its color and weight. This can be achieved by a multi-valued hash map.

mhashmap = o3c.HashMap(capacity,

key\_dtype=o3c.int32,

key\_element\_shape=(3,),

value\_dtypes=(o3c.uint8, o3c.float32),

value\_element\_shapes=((3,), (1,)),

device=device)

voxel\_coords = o3c.Tensor([[0, 1, 0], [-1, 2, 3], [3, 4, 1]],

dtype=o3c.int32,

device=device)

voxel\_colors = o3c.Tensor([[0, 255, 0], [255, 255, 0], [255, 0, 0]],

dtype=o3c.uint8,

device=device)

voxel\_weights = o3c.Tensor([[0.9], [0.1], [0.3]],

dtype=o3c.float32,

device=device)

mhashmap.insert(voxel\_coords, (voxel\_colors, voxel\_weights))

We can then query and access by indices with a slightly different routine:

query\_coords = o3c.Tensor([[0, 1, 0]], dtype=o3c.int32, device=device)

buf\_indices, masks = mhashmap.find(query\_coords)

valid\_keys = query\_coords[masks]

buf\_indices = buf\_indices[masks].to(o3c.int64)

valid\_colors = mhashmap.value\_tensor(0)[buf\_indices]

valid\_weights = mhashmap.value\_tensor(1)[buf\_indices]

print('found coordinates:\n', valid\_keys)

print('found colors:\n', valid\_colors)

print('found weights:\n', valid\_weights)

#### Hash set

Hash set is a simplified hash map where we do not care about the values. It preserves most of the operations in a hash map, and could be useful for removing duplicates.

hashset = o3c.HashSet(capacity,

key\_dtype=o3c.int64,

key\_element\_shape=(1,),

device=device)

keys = o3c.Tensor([1, 3, 5, 7, 5, 3, 1], dtype=o3c.int64,

device=device).reshape((-1, 1))

hashset.insert(keys)

keys = o3c.Tensor([5, 7, 9, 11], dtype=o3c.int64, device=device).reshape(

(-1, 1))

hashset.insert(keys)

def print\_active\_keys(hashset):

active\_buf\_indices = hashset.active\_buf\_indices().to(o3c.int64)

active\_keys = hashset.key\_tensor()[active\_buf\_indices]

print('active keys:\n', active\_keys)

print\_active\_keys(hashset)

## Geometry

### Point Cloud

<https://www.open3d.org/docs/release/tutorial/geometry/pointcloud.html>

#### Visualize point cloud

The first part of the tutorial reads a point cloud and visualizes it.

print("Load a ply point cloud, print it, and render it")

ply\_point\_cloud = o3d.data.PLYPointCloud()

pcd = o3d.io.read\_point\_cloud(ply\_point\_cloud.path)

print(pcd)

print(np.asarray(pcd.points))

o3d.visualization.draw\_geometries([pcd],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

***read\_point\_cloud*** reads a point cloud from a file.

***draw\_geometries*** visualizes the point cloud.

#### Voxel downsampling

Voxel downsampling uses a regular voxel grid to create a uniformly downsampled point cloud from an input point cloud. It is often used as a pre-processing step for many point cloud processing tasks. The algorithm operates in two steps:

1. Points are bucketed into voxels.
2. Each occupied voxel generates exactly one point by averaging all points inside.

downpcd = pcd.voxel\_down\_sample(voxel\_size=0.05)

#### Vertex normal estimation

Another basic operation for point cloud is **point normal estimation**.

downpcd.estimate\_normals(

search\_param=o3d.geometry.KDTreeSearchParamHybrid(radius=0.1, max\_nn=30))

#### Access estimated vertex normal

**Estimated normal vectors** can be retrieved from the normals variable of downpcd.

print(downpcd.normals[0])

#### Crop point cloud

print("Load a polygon volume and use it to crop the original point cloud")

demo\_crop\_data = o3d.data.DemoCropPointCloud()

pcd = o3d.io.read\_point\_cloud(demo\_crop\_data.point\_cloud\_path)

vol = o3d.visualization.read\_selection\_polygon\_volume(demo\_crop\_data.cropped\_json\_path)

chair = vol.crop\_point\_cloud(pcd)

#### Paint point cloud

print("Paint chair")

chair.paint\_uniform\_color([1, 0.706, 0])

#### Point cloud distance

Open3D provides the method **compute\_point\_cloud\_distance** to compute the distance from a source point cloud to a target point cloud. I.e., it computes for each point in the source point cloud the distance to the closest point in the target point cloud.

In the example below we use the function to compute the difference between two point clouds.

# Load data

demo\_crop\_data = o3d.data.DemoCropPointCloud()

pcd = o3d.io.read\_point\_cloud(demo\_crop\_data.point\_cloud\_path)

vol = o3d.visualization.read\_selection\_polygon\_volume(demo\_crop\_data.cropped\_json\_path)

chair = vol.crop\_point\_cloud(pcd)

dists = pcd.compute\_point\_cloud\_distance(chair)

dists = np.asarray(dists)

ind = np.where(dists > 0.01)[0]

pcd\_without\_chair = pcd.select\_by\_index(ind)

o3d.visualization.draw\_geometries([pcd\_without\_chair],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

#### Bounding volumes

The PointCloud geometry type has bounding volumes as all other geometry types in Open3D. Currently, Open3D implements an AxisAlignedBoundingBox and an OrientedBoundingBox that can also be used to crop the geometry.

aabb = chair.get\_axis\_aligned\_bounding\_box()

aabb.color = (1, 0, 0)

obb = chair.get\_oriented\_bounding\_box()

obb.color = (0, 1, 0)

o3d.visualization.draw\_geometries([chair, aabb, obb],

zoom=0.7,

front=[0.5439, -0.2333, -0.8060],

lookat=[2.4615, 2.1331, 1.338],

up=[-0.1781, -0.9708, 0.1608])

#### Convex hull

The convex hull of a point cloud is the smallest convex set that contains all points. Open3D contains the method **compute\_convex\_hull** that computes the convex hull of a point cloud. The implementation is based on **Qhull**.

bunny = o3d.data.BunnyMesh()

mesh = o3d.io.read\_triangle\_mesh(bunny.path)

mesh.compute\_vertex\_normals()

pcl = mesh.sample\_points\_poisson\_disk(number\_of\_points=2000)

hull, \_ = pcl.compute\_convex\_hull()

hull\_ls = o3d.geometry.LineSet.create\_from\_triangle\_mesh(hull)

hull\_ls.paint\_uniform\_color((1, 0, 0))

o3d.visualization.draw\_geometries([pcl, hull\_ls])

#### DBSCAN clustering

Given a point cloud from e.g. a depth sensor we want to group local point cloud clusters together. For this purpose, we can use clustering algorithms.

Open3D implements DBSCAN [[Ester1996]](https://www.open3d.org/docs/release/tutorial/reference.html#Ester1996) that is a density based clustering algorithm. The algorithm is implemented in **cluster\_dbscan** and requires two parameters: eps defines the distance to neighbors in a cluster and min\_points defines the minimum number of points required to form a cluster.

The function returns labels, where the label -1 indicates noise.

ply\_point\_cloud = o3d.data.PLYPointCloud()

pcd = o3d.io.read\_point\_cloud(ply\_point\_cloud.path)

with o3d.utility.VerbosityContextManager(

o3d.utility.VerbosityLevel.Debug) **as** cm:

labels = np.array(

pcd.cluster\_dbscan(eps=0.02, min\_points=10, print\_progress=**True**))

max\_label = labels.max()

#### Plane segmentation

Open3D also supports segmententation of geometric primitives from point clouds using RANSAC. To find the plane with the largest support in the point cloud, we can use **segment\_plane**. The method has three arguments: distance\_threshold defines the maximum distance a point can have to an estimated plane to be considered an inlier, ransac\_n defines the number of points that are randomly sampled to estimate a plane, and num\_iterations defines how often a random plane is sampled and verified.

pcd\_point\_cloud = o3d.data.PCDPointCloud()

pcd = o3d.io.read\_point\_cloud(pcd\_point\_cloud.path)

plane\_model, inliers = pcd.segment\_plane(distance\_threshold=0.01,

ransac\_n=3,

num\_iterations=1000)

[a, b, c, d] = plane\_model

print(f"Plane equation: *{*a*:*.2f*}*x + *{*b*:*.2f*}*y + *{*c*:*.2f*}*z + *{*d*:*.2f*}* = 0")

#### Planar patch detection

In addition to finding the single plane with the largest support, Open3D includes an algorithm which uses a robust statistics-based approach for planar patch detectio

dataset = o3d.data.PCDPointCloud()

pcd = o3d.io.read\_point\_cloud(dataset.path)

assert (pcd.has\_normals())

# using all defaults

oboxes = pcd.detect\_planar\_patches(

normal\_variance\_threshold\_deg=60,

coplanarity\_deg=75,

outlier\_ratio=0.75,

min\_plane\_edge\_length=0,

min\_num\_points=0,

search\_param=o3d.geometry.KDTreeSearchParamKNN(knn=30))

#### Hidden point removal

Imagine you want to render a point cloud from a given view point, but points from the background leak into the foreground because they are not occluded by other points. For this purpose we can apply a hidden point removal algorithm.

\_, pt\_map = pcd.hidden\_point\_removal(camera, radius)

### Mesh

<https://www.open3d.org/docs/release/tutorial/geometry/mesh.html>

Open3D has a data structure for 3D triangle meshes called **TriangleMesh**. The code below shows how to read a triangle mesh from a ply file and print its vertices and triangles.

print("Testing mesh in Open3D...")

armadillo\_mesh = o3d.data.ArmadilloMesh()

mesh = o3d.io.read\_triangle\_mesh(armadillo\_mesh.path)

knot\_mesh = o3d.data.KnotMesh()

mesh = o3d.io.read\_triangle\_mesh(knot\_mesh.path)

print(mesh)

print('Vertices:')

print(np.asarray(mesh.vertices))

print('Triangles:')

print(np.asarray(mesh.triangles))

The TriangleMesh class has a few data fields such as vertices and triangles.

#### Visualize a 3D mesh

print("Try to render a mesh with normals (exist: " +

str(mesh.has\_vertex\_normals()) + ") and colors (exist: " +

str(mesh.has\_vertex\_colors()) + ")")

o3d.visualization.draw\_geometries([mesh])

#### Surface normal estimation

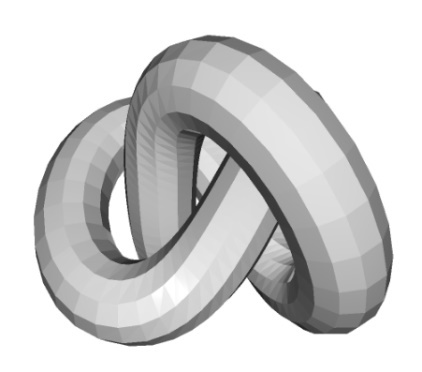
Let’s draw the mesh with surface normals.

print("Computing normal and rendering it.")

mesh.compute\_vertex\_normals()

print(np.asarray(mesh.triangle\_normals))

o3d.visualization.draw\_geometries([mesh])



#### Crop mesh

We remove half of the surface by directly operating on the triangle and triangle\_normals data fields of the mesh. This is done via numpy.

print("We make a partial mesh of only the first half triangles.")

mesh1 = copy.deepcopy(mesh)

mesh1.triangles = o3d.utility.Vector3iVector(

np.asarray(mesh1.triangles)[:len(mesh1.triangles) // 2, :])

mesh1.triangle\_normals = o3d.utility.Vector3dVector(

np.asarray(mesh1.triangle\_normals)

[:len(mesh1.triangle\_normals) // 2, :])

print(mesh1.triangles)

o3d.visualization.draw\_geometries([mesh1])

#### Paint mesh

**paint\_uniform\_color** paints the mesh with a uniform color.

print("Painting the mesh")

mesh1.paint\_uniform\_color([1, 0.706, 0])

o3d.visualization.draw\_geometries([mesh1])

#### Mesh properties

A triangle mesh has several properties that can be tested with Open3D. One important property is the manifold property, where we can test the triangle mesh if it is edge manifold is\_edge\_manifold and if it is is\_vertex\_manifold. A triangle mesh is edge manifold, if each edge is bounding either one or two triangles.

def check\_properties(name, mesh):

mesh.compute\_vertex\_normals()

edge\_manifold = mesh.is\_edge\_manifold(allow\_boundary\_edges=**True**)

edge\_manifold\_boundary = mesh.is\_edge\_manifold(allow\_boundary\_edges=**False**)

vertex\_manifold = mesh.is\_vertex\_manifold()

self\_intersecting = mesh.is\_self\_intersecting()

watertight = mesh.is\_watertight()

orientable = mesh.is\_orientable()

#### Mesh filtering

##### Average filter

This filter can be used to denoise meshes as demonstrated in the code below. The parameter number\_of\_iterations in the function **filter\_smooth\_simple** defines the how often the filter is applied to the mesh.

##### Laplacian

The filter is implemented in **filter\_smooth\_laplacian** and has the parameters number\_of\_iterations and lambda.

##### Taubin filter

The problem with the average and Laplacian filter is that they lead to a shrinkage of the triangle mesh. [Taubin1995] showed that the application of two Laplacian filters with different parameters can prevent the mesh shrinkage. The filter is implemented in **filter\_smooth\_taubin**.

#### Sampling

Open3D includes functions to sample point clouds from a triangle mesh. The simplest method is sample\_points\_uniformly that uniformly samples points from the 3D surface based on the triangle area. The parameter number\_of\_points defines how many points are sampled from the triangle surface.

mesh = o3d.geometry.TriangleMesh.create\_sphere()

mesh.compute\_vertex\_normals()

o3d.visualization.draw\_geometries([mesh])

pcd = mesh.sample\_points\_uniformly(number\_of\_points=500)

o3d.visualization.draw\_geometries([pcd])

#### Mesh subdivision

In mesh subdivision we divide each triangle into a number of smaller triangles. In the simplest case, we compute the midpoint of each side per triangle and divide the triangle into four smaller triangles. This is implemented in the **subdivide\_midpoint** function.

The 3D surface and area stays the same, but the number of vertices and triangles increases. The parameter number\_of\_iterations defines how many times this process should be repeated.

mesh = o3d.geometry.TriangleMesh.create\_box()

mesh.compute\_vertex\_normals()

print(

f'The mesh has {len(mesh.vertices)} vertices and {len(mesh.triangles)} triangles'

)

o3d.visualization.draw\_geometries([mesh], zoom=0.8, mesh\_show\_wireframe=True)

mesh = mesh.subdivide\_midpoint(number\_of\_iterations=1)

print(

f'After subdivision it has {len(mesh.vertices)} vertices and {len(mesh.triangles)} triangles'

)

o3d.visualization.draw\_geometries([mesh], zoom=0.8, mesh\_show\_wireframe=True)

#### Mesh simplification

Sometimes we want to represent a high-resolution mesh with fewer triangles and vertices, but the low-resolution mesh should still be close to the high-resolution mesh.

##### Vertex clustering

The vertex clustering method pools all vertices that fall into a voxel of a given size to a single vertex. The method is implemented in **simplify\_vertex\_clustering**

mesh\_smp = mesh\_in.simplify\_vertex\_clustering(

voxel\_size=voxel\_size,

contraction=o3d.geometry.SimplificationContraction.Average)

##### Mesh decimation

Another category of mesh simplification methods is mesh decimation that operates in incremental steps. We select a single triangle that minimizes an error metric and removes it. This is repeated until a required number of triangles is achieved.

Open3D implements **simplify\_quadric\_decimation**that minimizes error quadrics (distances to neighboring planes). The parameter target\_number\_of\_triangles defines the stopping criteria of the decimation algorithm.

mesh\_smp = mesh\_in.simplify\_quadric\_decimation(target\_number\_of\_triangles=1700)

#### Connected components

The result of various reconstruction methods. Open3D implements a connected components algorithm cluster\_connected\_triangles that assigns each triangle to a cluster of connected triangles. It returns for each triangle the index of the cluster in triangle\_clusters, and per cluster the number of triangles in cluster\_n\_triangles and the surface area of the cluster in cluster\_area.

### RGBD images

Open3D has a data structure for images. It supports various functions such as read\_image, write\_image, filter\_image and draw\_geometries. An Open3D Image can be directly converted to/from a numpy array.

An Open3D RGBDImage is composed of two images, RGBDImage.depth and RGBDImage.color. We require the two images to be registered into the same camera frame and have the same resolution.

#### Redwood dataset

In this section we show how to read and visualize an RGBDImage from the **Redwood** dataset [Choi2015].

The Redwood format stored depth in a 16-bit single channel image.

#### SUN dataset

#### NYU dataset

### KDTree

Open3D uses FLANN to build KDTrees for fast retrieval of nearest neighbors.

#### Build KDTree from point cloud

The code below reads a point cloud and builds a KDTree. This is a preprocessing step for the following nearest neighbor queries.

print("Testing kdtree in Open3D...")

print("Load a point cloud and paint it gray.")

sample\_pcd\_data = o3d.data.PCDPointCloud()

pcd = o3d.io.read\_point\_cloud(sample\_pcd\_data.path)

pcd.paint\_uniform\_color([0.5, 0.5, 0.5])

pcd\_tree = o3d.geometry.KDTreeFlann(pcd)

#### Find neighboring points

##### Using search\_knn\_vector\_3d

The function **search\_knn\_vector\_**3d returns a list of indices of the k nearest neighbors of the anchor point.

print("Find its 200 nearest neighbors, and paint them blue.")

[k, idx, \_] = pcd\_tree.search\_knn\_vector\_3d(pcd.points[1500], 200)

np.asarray(pcd.colors)[idx[1:], :] = [0, 0, 1]

##### Using search\_radius\_vector\_3d

Similarly, we can use **search\_radius\_vector\_3d** to query all points with distances to the anchor point less than a given radius.

print("Find its neighbors with distance less than 0.2, and paint them green.")

[k, idx, \_] = pcd\_tree.search\_radius\_vector\_3d(pcd.points[1500], 0.2)

np.asarray(pcd.colors)[idx[1:], :] = [0, 1, 0]

### File IO

This tutorial shows how basic data structures are read and written by Open3D.

#### Point cloud

The code below reads and writes a point cloud.

print("Testing IO for point cloud ...")

sample\_pcd\_data = o3d.data.PCDPointCloud()

pcd = o3d.io.read\_point\_cloud(sample\_pcd\_data.path)

print(pcd)

o3d.io.write\_point\_cloud("copy\_of\_fragment.pcd", pcd)

#### Mesh

The code below reads and writes a mesh.

print("Testing IO for meshes ...")

knot\_data = o3d.data.KnotMesh()

mesh = o3d.io.read\_triangle\_mesh(knot\_data.path)

print(mesh)

o3d.io.write\_triangle\_mesh("copy\_of\_knot.ply", mesh)

#### Image

The code below reads and writes an image.

print("Testing IO for images ...")

image\_data = o3d.data.JuneauImage()

img = o3d.io.read\_image(image\_data.path)

print(img)

o3d.io.write\_image("copy\_of\_Juneau.jpg", img)

### Point cloud outlier removal

When collecting data from scanning devices, the resulting point cloud tends to contain noise and artifacts that one would like to remove.

#### Prepare input data

A point cloud is loaded and downsampled

print("Load a ply point cloud, print it, and render it")

sample\_pcd\_data = o3d.data.PCDPointCloud()

pcd = o3d.io.read\_point\_cloud(sample\_pcd\_data.path)

o3d.visualization.draw\_geometries([pcd],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

print("Downsample the point cloud with a voxel of 0.02")

voxel\_down\_pcd = pcd.voxel\_down\_sample(voxel\_size=0.02)

o3d.visualization.draw\_geometries([voxel\_down\_pcd],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

Alternatively, use **uniform\_down\_sample** to downsample the point cloud by collecting every n-th points.

print("Every 5th points are selected")

uni\_down\_pcd = pcd.uniform\_down\_sample(every\_k\_points=5)

o3d.visualization.draw\_geometries([uni\_down\_pcd],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

#### Select down sample

The following helper function uses select\_by\_index, which takes a binary mask to output only the selected points. The selected points and the non-selected points are visualized.

inlier\_cloud = cloud.select\_by\_index(ind)

#### Statistical outlier removal

**statistical\_outlier\_removal** removes points that are further away from their neighbors compared to the average for the point cloud.

cl, ind = voxel\_down\_pcd.remove\_statistical\_outlier(nb\_neighbors=20,

std\_ratio=2.0)

### Voxelization

 The voxel grid is another geometry type in 3D that is defined on a regular 3D grid

#### Claude - Q: What is a voxel grid?

A voxel grid is a fundamental 3D data structure in computer vision that represents three-dimensional space as a regular, cubic lattice of volume elements called "voxels." Think of it as the 3D equivalent of how pixels work in 2D images - where pixels are picture elements that make up a flat image, voxels are volume elements that make up a 3D representation.

Let me help you build an understanding of this concept step by step, starting with the basics and moving toward more complex applications.

Understanding the Basic Structure

Imagine you're trying to represent a 3D object digitally. One approach is to divide the surrounding 3D space into a regular grid of tiny cubes, similar to how you might use building blocks or Minecraft blocks to construct something. Each of these cubes is a voxel, and it can either be "occupied" (containing part of the object) or "empty" (containing just air or background).

The voxel grid extends in three dimensions with coordinates typically labeled as (x, y, z). If you have a grid that's 64×64×64, you have 262,144 individual voxels, each representing a small cubic region of space. The resolution of your grid determines how fine the details you can capture - smaller voxels mean higher resolution but also more memory and computational requirements.

How Voxel Grids Store Information

Each voxel can store different types of information depending on the application. In the simplest case, a voxel might just contain a binary value indicating occupancy (1 for occupied, 0 for empty). However, voxels can also store much richer information such as color values (RGB), density measurements, surface normals, or even semantic labels indicating what type of object or material occupies that space.

This flexibility makes voxel grids particularly powerful for representing complex 3D scenes where you need to capture not just shape, but also appearance and material properties throughout the volume.

Key Applications in Computer Vision

Voxel grids serve several important purposes in computer vision systems. In 3D object detection, they provide a natural way to represent the world that autonomous vehicles or robots need to navigate. When a self-driving car's lidar sensor captures millions of 3D points, these can be organized into a voxel grid to create a structured representation that neural networks can process efficiently.

For 3D reconstruction tasks, voxel grids offer a way to combine information from multiple camera viewpoints or depth sensors. As you gather evidence about which parts of space are occupied from different angles, you can update the corresponding voxels to build up a complete 3D model of a scene or object.

Medical imaging represents another crucial application area. CT scans and MRI data are naturally voxel-based, where each voxel represents the density or magnetic properties of tissue at that location. Computer vision algorithms can then analyze these voxel grids to detect tumors, segment organs, or plan surgical procedures.

Advantages and Trade-offs

The regular, grid-based structure of voxels brings several computational advantages. It's straightforward to implement 3D convolutions over voxel grids, making them well-suited for deep learning approaches. The uniform spacing also simplifies many geometric operations and spatial queries.

“””””””””””End of Claud explanation “”””””””””””””””””””””””””””””””””

From triangle mesh

Open3D provides the method create\_from\_triangle\_mesh that creates a voxel grid from a triangle mesh.

bunny = o3d.data.BunnyMesh()

mesh = o3d.io.read\_triangle\_mesh(bunny.path)

# fit to unit cube

mesh.scale(1 / np.max(mesh.get\_max\_bound() - mesh.get\_min\_bound()),

center=mesh.get\_center())

voxel\_grid = o3d.geometry.VoxelGrid.create\_from\_triangle\_mesh(mesh,

voxel\_size=0.05)

From point cloud

The voxel grid can also be created from a point cloud using the method create\_from\_point\_cloud. A voxel is occupied if at least one point of the point cloud is within the voxel. The argument voxel\_size defines the resolution of the voxel grid. By default, the color of the voxel is the average of all the points within the voxel.

The argument pooling\_mode can be changed to determine the color by average, min, max or sum value of the points, e.g. with o3d.geometry.VoxelGrid.VoxelPoolingMode.MIN.

voxel\_grid = o3d.geometry.VoxelGrid.create\_from\_point\_cloud(pcd, voxel\_size=0.05)

Voxel carving

The methods create\_from\_point\_cloud and create\_from\_triangle\_mesh create occupied voxels only on the surface of the geometry. It is however possible to carve a voxel grid from a number of depth maps or silhouettes. Open3D provides the methods carve\_depth\_map and carve\_silhouette for voxel carving.

# depth map carving method

if use\_depth:

voxel\_carving.carve\_depth\_map(o3d.geometry.Image(depth), param)

else:

voxel\_carving.carve\_silhouette(o3d.geometry.Image(depth), param)

### Octree

An **octree** is a tree data structure where each internal node has eight children. Octrees are commonly used for spatial partitioning of 3D point clouds.

Octrees are a useful description of 3D space and can be used to quickly find nearby points.

Open3D has the geometry type Octree that can be used to create, search, and traverse octrees with a user-specified maximum tree depth, max\_depth.

#### Claude - Q: What is an Octree

An octree is a clever tree-based data structure that solves one of the biggest problems with voxel grids: wasted space and computational resources.

The Problem That Octrees Solve

Remember how we discussed that voxel grids can be memory-intensive because they store information for every single voxel, even empty ones?

Think of an octree as a smart way to say "I only care about the interesting parts of my 3D space, and I want to ignore all the boring empty regions."

“”””””””””””””””””” end of Claude explanation “”””””””””””””””””””””””

From point cloud

An octree can be constructed from a point cloud using the method convert\_from\_point\_cloud.

octree = o3d.geometry.Octree(max\_depth=4)

octree.convert\_from\_point\_cloud(pcd, size\_expand=0.01)

From voxel grid

An octree can also be constructed from an Open3D VoxelGrid geometry using the method create\_from\_voxel\_grid.

Traversal

An octree can be traversed which can be useful for searching or processing subsections of 3D geometry. By providing the traverse method with a callback, each time a node (internal or leaf) is visited, additional processing can be performed.

### Surface reconstruction

In many scenarios we want to generate a dense 3D geometry, i.e., a triangle mesh. However, from a multi-view stereo method, or a depth sensor we only obtain an unstructured point cloud.

To get a triangle mesh from this unstructured input we need to perform surface reconstruction.

In the literature there exists a couple of methods and Open3D currently implements the following:

* Alpha shapes
* Ball pivoting
* Poisson surface reconstruction

### Transformation

The geometry types of Open3D have a number of transformation methods. In this tutorial we show how to use translate, rotate, scale, and transform.

General transformation

Open3D also supports a general transformation defined by a  homogeneous transformation matrix using the method transform.

### Mesh deformation

If we want to deform a triangle mesh according to a small number of constraints, we can use mesh deformation algorithms. Open3D implements the as-rigid-as-possible method by [[SorkineAndAlexa2007]](https://www.open3d.org/docs/release/tutorial/reference.html#SorkineAndAlexa2007)

### Intrinsic shape signatures (ISS)

In this tutorial we will show how to detect the **ISS** Keypoints of a 3D shape. The implementation is based on the keypoint detection modules proposed in Yu Zhong , “Intrinsic Shape Signatures: A Shape Descriptor for 3D Object Recognition”, 2009.

#### Claude – Q: What are Intrinsic shape signatures (ISS) ?

The Core Philosophy Behind Intrinsic Signatures

Think about how you recognize a friend's face. You don't identify them by measuring distances to walls or noting their position in a room. Instead, you recognize intrinsic features like the relationship between their eyes, nose, and mouth. Intrinsic Shape Signatures apply this same principle to 3D geometry. They capture properties that are inherent to the shape itself, independent of where it sits in space, how it's oriented, or even how it might be stretched or deformed.

### Ray Casting

The RaycastingScene class in Open3D provides basic ray casting functionality.

In this tutorial we show how to create a scene and do ray intersection tests.

You can also use RaycastingScene to create a virtual point cloud from a mesh, such as from a CAD model.

#### Claude – Q: What is Ray Casting?

Ray casting is one of the most fundamental and intuitive concepts in computer graphics and computer vision. Think of it as the digital equivalent of how light works in the real world, but with the ability to trace light rays backwards from your eye to understand what you're seeing.

Building Your Mental Model

Let me start with a simple analogy that will help you understand the core concept. Imagine you're standing in a dark room with a flashlight, and you want to understand the 3D structure around you. You could shine your flashlight in different directions and see what it illuminates. Ray casting works similarly, but instead of shining light outward, we trace imaginary rays from our viewpoint (like a camera or your eye) into the 3D scene to discover what objects exist and where they're located.

### Distance Queries

The RaycastingScene class in Open3D provides a set of distance queries, which can be used to convert triangle meshes into implicit functions, query the distance to the surface or determine if a point is inside a mesh.

#### Claude – Q: What do you know about Distance Queries in the context of RaycastingScene in Open3D?

Understanding Open3D's RaycastingScene Context

Open3D's RaycastingScene represents a sophisticated implementation of the ray casting concepts we explored earlier, but with a specific focus on efficient spatial queries and geometric analysis. Think of it as a highly optimized 3D spatial database that's been specifically designed to answer questions like "what's the closest surface to this point?" or "how far is it from here to the nearest obstacle?"

The RaycastingScene in Open3D essentially takes your 3D geometry, whether it's triangle meshes, point clouds, or other geometric primitives, and organizes it into an internal spatial acceleration structure. This structure enables incredibly fast distance computations that would be prohibitively slow if you had to check every triangle or point individually.

When you create a RaycastingScene, you're essentially telling Open3D "I'm going to be asking lots of spatial questions about this geometry, so please organize it in a way that makes those questions fast to answer."

### UV Maps

UV mapping is the 3D modeling process of projecting a 2D image to a 3D model’s surface for texture mapping. The letters “U” and “V” denote the axes of the 2D texture because “X”, “Y”, and “Z” are already used to denote the axes of the 3D object in model space. UV texturing permits polygons that make up a 3D object to be painted with color (and other surface attributes) from an ordinary image. The image is called a UV texture map.

#### Calude - Q : What are UV Maps?

Think of it as the bridge between the 3D world of computer graphics and the 2D world of traditional image editing and texturing.

Building Your Mental Model with Everyday Examples

Let me start with an analogy that will make everything clear. Imagine you have a globe of the Earth, and you want to create a flat world map that you can print on paper. This is exactly the challenge that cartographers have faced for centuries. How do you take something that's curved in 3D space and flatten it onto a 2D surface while preserving as much information as possible?

When cartographers create world maps, they use various projection techniques. The Mercator projection, for example, unwraps the globe as if it were a cylinder, creating those familiar rectangular world maps where Greenland appears much larger than it actually is. This process of taking 3D surface information and mapping it to 2D coordinates is precisely what UV mapping does for 3D computer graphics.

Now imagine you want to paint details onto that globe, but instead of painting directly on the curved surface, you want to paint on a flat canvas and then somehow wrap that flat artwork onto the sphere. You'd need a systematic way to know which part of your flat painting corresponds to which part of the globe's surface. UV mapping provides exactly this systematic correspondence.

Understanding the UV Coordinate System

Let's build your understanding of the coordinate system itself. In 3D graphics, we typically use X, Y, and Z coordinates to describe positions in three-dimensional space. UV coordinates work differently, they describe positions on a flat, two-dimensional surface using just two values: U and V.

### Python Interface

Getting started

This tutorial shows how to import the open3d module and use it to load and inspect a point cloud.

import open3d as o3d

Using external Open3D-ML in /home/runner/work/Open3D/Open3D/Open3D-ML

**Note:** Depending on the environment, the name of the Python library may not be open3d.so. Regardless of the file name, import open3d should work.

sample\_pcd\_data = o3d.data.PCDPointCloud()

pcd = o3d.io.read\_point\_cloud(sample\_pcd\_data.path)

print(pcd)

This imports the read\_point\_cloud function from the open3d module. It reads a point cloud file and returns an instance of the **PointCloud** class. print(pcd) prints some brief information about the point cloud.

#### Using built-in help function

Browse Open3D

help(open3d) prints a description of the open3d module.

Description of a class in Open3D

help(open3d.PointCloud) provides a description of the PointCloud class.

Description of a function in Open3D

help(open3d.read\_point\_cloud) provides a description of the input arguments and return type of the read\_point\_cloud function.

### Working with NumPy

All data structures in Open3D are natively compatible with a NumPy buffer.

From NumPy to open3d.PointCloud

Open3D provides conversion from a NumPy matrix to a vector of 3D vectors. By using Vector3dVector, a NumPy matrix can be directly assigned to open3d.PointCloud.points.

## Geometry (Tensor)

### PointCloud

This tutorial demonstrates basic usage of a point cloud.

#### PointCloud creation

# Create a empty point cloud on CPU.

pcd = o3d.t.geometry.PointCloud()

print(pcd, "\n")

# To create a point cloud on CUDA, specify the device.

# pcd = o3d.t.geometry.PointCloud(o3c.Device("cuda:0"))

# Create a point cloud from open3d tensor with dtype of float32.

pcd = o3d.t.geometry.PointCloud(o3c.Tensor([[0, 0, 0], [1, 1, 1]], o3c.float32))

print(pcd, "\n")

# Create a point cloud from open3d tensor with dtype of float64.

pcd = o3d.t.geometry.PointCloud(o3c.Tensor([[0, 0, 0], [1, 1, 1]], o3c.float64))

print(pcd, "\n")

# Create a point cloud from numpy array. The array will be copied.

pcd = o3d.t.geometry.PointCloud(

np.array([[0, 0, 0], [1, 1, 1]], dtype=np.float32))

print(pcd, "\n")

# Create a point cloud from python list.

pcd = o3d.t.geometry.PointCloud([[0., 0., 0.], [1., 1., 1.]])

print(pcd, "\n")

# Error creation. The point cloud must have shape of (N, 3).

try:

pcd = o3d.t.geometry.PointCloud(o3c.Tensor([0, 0, 0, 0], o3c.float32))

except:

print(f"Error creation. The point cloud must have shape of (N, 3).")

#### Point cloud attributes setter and getter

pcd = o3d.t.geometry.PointCloud(o3c.Tensor([[0, 0, 0], [1, 1, 1]], o3c.float32))

# Set attributes.

pcd.point.normals = o3c.Tensor([[0, 0, 1], [0, 0, 1]], o3c.float32)

pcd.point.colors = o3c.Tensor([[1, 0, 0], [0, 1, 0]], o3c.float32)

pcd.point.labels = o3c.Tensor([0, 1], o3c.int64)

print(pcd, "**\n**")

# Set by numpy array or python list.

pcd.point.normals = np.array([[0, 0, 1], [0, 0, 1]], dtype=np.float32)

pcd.point.intensity = [0.4, 0.4]

print(pcd, "**\n**")

# Get attributes.

posisions = pcd.point.positions

print("posisions: ")

print(posisions, "**\n**")

labels = pcd.point.labels

print("labels: ")

print(labels, "")

#### Visualize point cloud

print("Load a ply point cloud, print it, and render it")

ply\_point\_cloud = o3d.data.PLYPointCloud()

pcd = o3d.t.io.read\_point\_cloud(ply\_point\_cloud.path)

print(pcd)

o3d.visualization.draw\_geometries([pcd.to\_legacy()],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

#### Downsampling

print("Downsample the point cloud with a voxel of 0.03")

downpcd = pcd.voxel\_down\_sample(voxel\_size=0.03)

o3d.visualization.draw\_geometries([downpcd.to\_legacy()],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

#### Farthest point downsampling

Farthest point sampling samples the point cloud by selecting the farthest point from the current selected point iteratively. It is used to sample the point cloud to a fixed number of points which holds the maximum geometrical information of the original point cloud.

print("Downsample the point cloud by selecting 5000 farthest points.")

downpcd\_farthest = pcd.farthest\_point\_down\_sample(5000)

o3d.visualization.draw\_geometries([downpcd\_farthest.to\_legacy()],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024])

#### Vertex normal estimation

Another basic operation for point cloud is normal estimation.

print("Recompute the normal of the downsampled point cloud using hybrid nearest neighbor search with 30 max\_nn and radius of 0.1m.")

downpcd.estimate\_normals(max\_nn=30, radius=0.1)

o3d.visualization.draw\_geometries([downpcd.to\_legacy()],

zoom=0.3412,

front=[0.4257, -0.2125, -0.8795],

lookat=[2.6172, 2.0475, 1.532],

up=[-0.0694, -0.9768, 0.2024],

point\_show\_normal=True)

#### Crop point cloud

print("Load a polygon volume and use it to crop the original point cloud")

demo\_crop\_data = o3d.data.DemoCropPointCloud()

pcd = o3d.t.io.read\_point\_cloud(demo\_crop\_data.point\_cloud\_path)

vol = o3d.visualization.read\_selection\_polygon\_volume(demo\_crop\_data.cropped\_json\_path)

chair = vol.crop\_point\_cloud(pcd.to\_legacy())

o3d.visualization.draw\_geometries([chair],

zoom=0.7,

front=[0.5439, -0.2333, -0.8060],

lookat=[2.4615, 2.1331, 1.338],

up=[-0.1781, -0.9708, 0.1608])

#### Paint point cloud

print("Paint point cloud.")

pcd.paint\_uniform\_color([1, 0.706, 0])

o3d.visualization.draw\_geometries([pcd.to\_legacy()],

zoom=0.7,

front=[0.5439, -0.2333, -0.8060],

lookat=[2.4615, 2.1331, 1.338],

up=[-0.1781, -0.9708, 0.1608])

#### Bounding volumes

The PointCloud geometry type has bounding volumes as all other geometry types in Open3D. Currently, Open3D implements an AxisAlignedBoundingBox and an OrientedBoundingBox that can also be used to crop the geometry.

#### Point cloud outlier removal

When collecting data from scanning devices, the resulting point cloud tends to contain noise and artifacts that one would like to remove. This demo below addresses the outlier removal features of Open3D.

Statistical outlier removal

statistical\_outlier\_removal removes points that are further away from their neighbors compared to the average for the point cloud

Radius outlier removal

radius\_outlier\_removal removes points that have few neighbors in a given sphere around them

#### Convex hull

The convex hull of a point cloud is the smallest convex set that contains all points. Open3D contains the method compute\_convex\_hull that computes the convex hull of a point cloud.

#### DBSCAN clustering

Given a point cloud from e.g. a depth sensor we want to group local point cloud clusters together. For this purpose, we can use clustering algorithms. Open3D implements DBSCAN that is a density based clustering algorithm. The algorithm is implemented in cluster\_dbscan

#### Plane segmentation

Open3D also supports segmententation of geometric primitives from point clouds using RANSAC.

#### Hidden point removal

Imagine you want to render a point cloud from a given view point, but points from the background leak into the foreground because they are not occluded by other points. For this purpose we can apply a hidden point removal algorithm.

#### Boundary detection

Open3D implements the boundary detection algorithm inspired by PCL. The algorithm find the boundary points among a unordered point cloud by analyzing the angle among the normals of a point and its neighbors.

## Dataset

Open3D comes with a built-in dataset module for convenient access to commonly used example datasets. These datasets will be downloaded automatically from the internet.

import open3d as o3d

if \_\_name\_\_ == "\_\_main\_\_":

dataset = o3d.data.EaglePointCloud()

pcd = o3d.io.read\_point\_cloud(dataset.path)

o3d.visualization.draw(pcd)

* Datasets are downloaded can cached automatically. The default data root is ~/open3d\_data. Data will be downloaded to ~/open3d\_data/download and extracted to ~/open3d\_data/extract.

## Visualization

### Visualization

Function draw\_geometries

Open3D provides a convenient visualization function draw\_geometries which takes a list of geometry objects (PointCloud, TriangleMesh, or Image), and renders them together.

pcd = o3d.io.read\_point\_cloud(sample\_ply\_data.path)

o3d.visualization.draw\_geometries([pcd],

Geometry primitives

The code below generates a box, a sphere, and a cylinder using create\_box, create\_sphere, and create\_cylinder. The box is painted in red, the sphere is painted in blue, and the cylinder is painted in green. Normals are computed for all meshes to support Phong shading. We can even create a coordinate axis using create\_coordinate\_frame, with its origin point set at (-2, -2, -2).

print("Let's define some primitives")

mesh\_box = o3d.geometry.TriangleMesh.create\_box(width=1.0,

height=1.0,

depth=1.0)

mesh\_box.compute\_vertex\_normals()

mesh\_box.paint\_uniform\_color([0.9, 0.1, 0.1])

mesh\_sphere = o3d.geometry.TriangleMesh.create\_sphere(radius=1.0)

mesh\_sphere.compute\_vertex\_normals()

mesh\_sphere.paint\_uniform\_color([0.1, 0.1, 0.7])

mesh\_cylinder = o3d.geometry.TriangleMesh.create\_cylinder(radius=0.3,

height=4.0)

mesh\_cylinder.compute\_vertex\_normals()

mesh\_cylinder.paint\_uniform\_color([0.1, 0.9, 0.1])

mesh\_frame = o3d.geometry.TriangleMesh.create\_coordinate\_frame(

size=0.6, origin=[-2, -2, -2])

Draw line set

To draw lines, it is necessary to define a LineSet with a set of points and a set of edges (pairs of point indices). The example below creates custom points and edges (denoted as lines) to make a box.

print("Let's draw a box using o3d.geometry.LineSet.")

points = [

[0, 0, 0],

[1, 0, 0],

…

[1, 1, 1],

]

lines = [

[0, 1],

[0, 2],

…

[3, 7],

]

colors = [[1, 0, 0] for i in range(len(lines))]

line\_set = o3d.geometry.LineSet(

points=o3d.utility.Vector3dVector(points),

lines=o3d.utility.Vector2iVector(lines),

)

line\_set.colors = o3d.utility.Vector3dVector(colors)

o3d.visualization.draw\_geometries([line\_set])

### Customized visualization

This tutorial focuses on more advanced functionalities to customize the behavior of the visualizer window.

### Interactive visualization

This tutorial introduces user interaction features of the visualizer window provided by:-

1. open3d.visualization.draw\_geometries\_with\_editing
2. open3d.visualization.VisualizerWithEditing

### Non-blocking visualization

### Headless rendering

This tutorial explains how to render and save images from a terminal without any display device.

Install OSMesa

To generate a headless context, it is necessary to install OSMesa.

$ sudo apt-get install libosmesa6-dev

### CPU (Software) Rendering

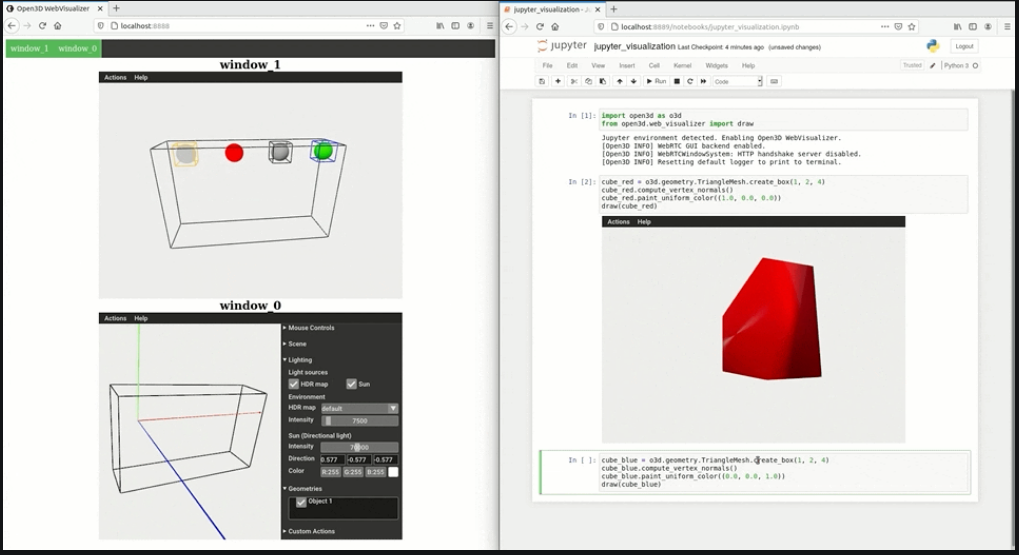
Open3D supports CPU rendering when a GPU is not available.

### Web visualizer and Jupyter

Open3D web visualizer enables advanced rendering and visualization of 3D models in modern browsers. You first start the visualization server on a local or remote machine, and then the 3D models can be viewed from any device with a modern browser. The web visualizer server and client communicate via the [WebRTC](https://en.wikipedia.org/wiki/WebRTC) (Web Real-Time Communication) protocol.

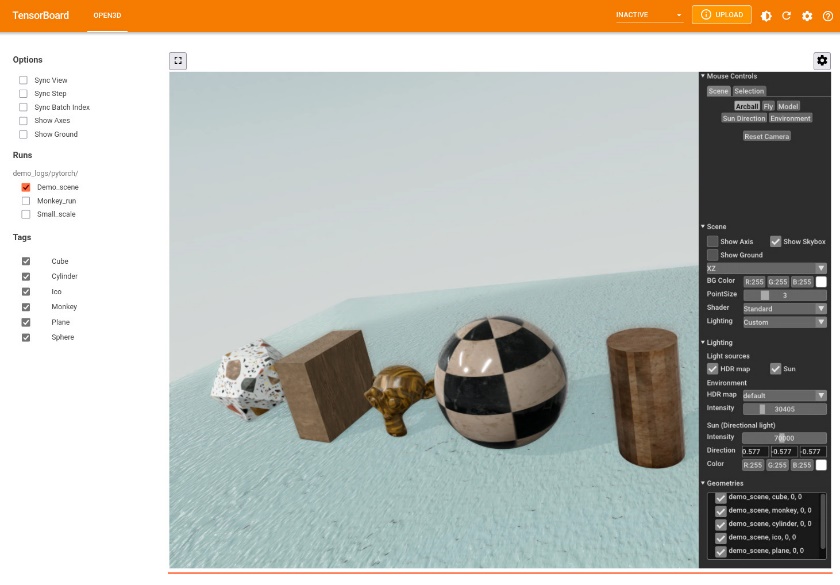
In this tutorial, we cover two common use cases:

* Standalone mode: Run Open3D web visualizer server as a standalone application and view the 3D models in a browser. It supports both C++ and Python. You’ll only need add one line to your code to enable the server.
* Jupyter mode: Run Open3D web visualizer as embedded Jupyter cells. It supports both the Python Jupyter notebook and Jupyter lab environments. You’ll need to modify your code to use the non-blocking APIs and wrappers.



### Open3D for TensorBoard

Now you can use Open3D within TensorBoard for interactive 3D visualization. Here are some of the exciting features:



#### Claude – Q: What is TensorBoard and how is it useful for Computer Vision?

TensorBoard is a visualization toolkit that transforms the abstract world of machine learning into something you can actually see and understand. Think of it as your microscope for peering into the inner workings of neural networks - particularly powerful when you're working with computer vision models that process images.

Let me walk you through this step by step, starting with the basics and then showing you why it becomes absolutely essential for computer vision work.

Understanding TensorBoard's Core Purpose

Imagine you're teaching a child to recognize different animals in photographs. You'd want to see their progress over time - are they getting better at distinguishing cats from dogs? Are they struggling with certain types of images? TensorBoard serves this exact role for neural networks, but instead of watching a child learn, you're monitoring how millions of artificial neurons adjust their connections to recognize patterns in images.

When you train a computer vision model, enormous amounts of data flow through the system every second. Without visualization, this would be like trying to understand a symphony by reading the sheet music as numbers rather than hearing the music. TensorBoard converts these numerical streams into visual stories that reveal what's happening during training.

Why Computer Vision Makes TensorBoard Especially Valuable

Computer vision presents unique challenges that make visualization tools like TensorBoard particularly crucial. Unlike working with simple numerical data, you're dealing with high-dimensional image data where problems can be subtle and hard to detect without proper visualization.

Consider what happens when your model processes a single image. The original image might contain millions of pixel values, and as it moves through different layers of your neural network, it gets transformed into increasingly abstract representations. Early layers might detect edges and basic shapes, while deeper layers recognize complex objects like faces or vehicles. Without TensorBoard, these transformations would be invisible to you.

Here's where TensorBoard becomes your window into this process. You can actually watch how your model's understanding evolves over time, see which types of images cause problems, and identify when your training process goes astray.

Tracking Training Dynamics in Computer Vision

Let's start with the most fundamental use case: monitoring how well your model learns over time. In computer vision tasks, you typically track several key metrics that tell the story of your model's progress.

The loss curves show you how your model's mistakes change during training. In computer vision, you might be tracking classification loss (how well the model predicts the correct category), localization loss (how accurately it identifies where objects are located), or reconstruction loss (how well it recreates input images). TensorBoard plots these values in real-time, letting you see patterns that would be impossible to catch by staring at numbers scrolling by in a terminal.

Think of it this way: if your validation loss suddenly starts increasing while training loss continues decreasing, you're witnessing overfitting in action. Your model is memorizing the training images rather than learning generalizable patterns. TensorBoard makes this visible as diverging lines on a graph, allowing you to stop training at the optimal point.

Visualizing What Your Model Actually Sees

Here's where TensorBoard becomes truly fascinating for computer vision work. You can log actual images at different stages of processing, creating a visual narrative of how your model transforms raw pixels into understanding.

For instance, imagine you're training a model to detect faces in photographs. You can log the original input images alongside the model's predictions, seeing exactly which faces it correctly identifies and which ones it misses. More importantly, you can visualize the intermediate representations - what does the image look like after passing through the first few layers? What patterns is the model focusing on?

This capability extends to feature maps, which show you what different parts of your neural network are responding to. You might discover that one layer consistently activates when it sees curved edges, while another responds to texture patterns. This insight helps you understand whether your model is learning meaningful visual features or just memorizing irrelevant details.

Debugging Common Computer Vision Problems

Computer vision models are notorious for learning unexpected patterns, and TensorBoard helps you catch these issues before they become serious problems. Let me give you a concrete example that illustrates this power.

Suppose you're training a model to classify medical images, but you notice the accuracy isn't improving as expected. By logging sample predictions in TensorBoard, you might discover that your model is actually learning to recognize the specific camera or scanning equipment used rather than the medical conditions themselves. The visualization would show you that images from different hospitals have systematically different predictions, even when they show the same condition.

Without TensorBoard, this type of dataset bias could remain hidden until your model fails completely in real-world deployment. The visual feedback helps you spot these patterns early and adjust your training data accordingly.

Advanced Visualization for Complex Architectures

Modern computer vision often involves sophisticated architectures like object detection networks, semantic segmentation models, or generative adversarial networks. TensorBoard provides specialized visualizations for these complex scenarios.

For object detection, you can visualize bounding boxes overlaid on images, watching how your model's predictions become more accurate over time. The early training epochs might show wildly incorrect boxes, but gradually you see them tighten around the correct objects.

## Pipelines

### ICP registration

This tutorial demonstrates the **ICP (Iterative Closest Point)** registration algorithm.

The input are two point clouds and an initial transformation that roughly aligns the source point cloud to the target point cloud. The output is a refined transformation that tightly aligns the two point clouds.

#### Claude - Q: What is ICP (Iterative Closest Point) and why should I care?

ICP (Iterative Closest Point) is a fundamental algorithm in computer vision and robotics that solves one of the most important problems in 3D geometry: how to align two similar point clouds when you don't know exactly how they're oriented relative to each other.

Let me build this up from the ground up so you can really understand what's happening here.

The Core Problem

Imagine you have two photographs of the same sculpture taken from slightly different angles, and you've converted each photo into a collection of 3D points representing the sculpture's surface. Now you want to figure out exactly how to rotate and translate one point cloud to perfectly align it with the other. This is called the "point cloud registration" problem, and it's surprisingly tricky because you don't know which point in the first cloud corresponds to which point in the second cloud.

How ICP Tackles This Challenge

ICP uses a clever iterative approach that alternates between two key steps. Think of it like a dance between making educated guesses and refining those guesses based on what you learn.

In the first step, the algorithm looks at each point in your first point cloud and asks: "What's the closest point to me in the second point cloud?" This creates temporary correspondences between the two sets of points.

Why Should You Care About ICP?

In robotics, it enables robots to build maps of their environment by aligning laser scans taken from different positions.

### Robust kernels

This tutorial demonstrates the use of robust kernels in the context of outlier rejection. For this particular tutorial, we will be using the ICP (Iterative Closest Point) registration algorithm as the *target* problem where we want to deal with outliers.

### Colored point cloud registration

This tutorial demonstrates an ICP variant that uses both geometry and color for registration.

### Global registration

Both ICP registration and Colored point cloud registration are known as local registration methods because they rely on a rough alignment as initialization. This tutorial shows another class of registration methods, known as global registration. This family of algorithms do not require an alignment for initialization. They usually produce less tight alignment results and are used as initialization of the local methods.

### Multiway registration

Multiway registration is the process of aligning multiple pieces of geometry in a global space. Typically, the input is a set of geometries (e.g., point clouds or RGBD images) . The output is a set of rigid transformations , so that the transformed point clouds  are aligned in the global space.

### RGBD integration

Open3D implements a scalable RGBD image integration algorithm.

#### Claude Q: What is RGBD integration and why should I care?

RGBD integration is a computer vision technique that combines regular color images (RGB) with depth information (D) to create detailed 3D reconstructions of scenes or objects. The "RGB" part captures the visual appearance - colors and textures - while the "D" part provides spatial depth measurements, typically from sensors like Microsoft Kinect, Intel RealSense cameras, or LiDAR.

The integration process involves aligning and fusing multiple RGBD frames taken from different viewpoints to build a coherent 3D model. This creates what's called a volumetric representation - essentially a digital 3D map that captures both the geometry and appearance of real-world environments.

Here's why you might care about RGBD integration:

**Robotics and Autonomous Systems**: Robots use RGBD integration for navigation, obstacle avoidance, and manipulation tasks.

### RGBD Odometry

An RGBD odometry finds the camera movement between two consecutive RGBD image pairs. The input are two instances of RGBDImage. The output is the motion in the form of a rigid body transformation.

Compute odometry from two RGBD image pairs

option = o3d.pipelines.odometry.OdometryOption()

odo\_init = np.identity(4)

print(option)

[success\_color\_term, trans\_color\_term,

info] = o3d.pipelines.odometry.compute\_rgbd\_odometry(

source\_rgbd\_image, target\_rgbd\_image, pinhole\_camera\_intrinsic, odo\_init,

o3d.pipelines.odometry.RGBDOdometryJacobianFromColorTerm(), option)

[success\_hybrid\_term, trans\_hybrid\_term,

info] = o3d.pipelines.odometry.compute\_rgbd\_odometry(

source\_rgbd\_image, target\_rgbd\_image, pinhole\_camera\_intrinsic, odo\_init,

o3d.pipelines.odometry.RGBDOdometryJacobianFromHybridTerm(), option)

### Color Map Optimization

As color and depth frames are not perfectly aligned, the texture mapping using color images is subject to results in blurred color map. Open3D provides color map optimization

## Pipelines (Tensor)

### ICP registration

This tutorial demonstrates the ICP (Iterative Closest Point) registration algorithm.

#### Claude – Q: What is ICP (Iterative Closest Point) registration? and why should we care?

Iterative Closest Point (ICP) registration is a fundamental algorithm used to align two point clouds or geometric datasets by finding the optimal transformation (rotation and translation) that minimizes the distance between corresponding points.

Why ICP Matters

**3D Computer Vision and Robotics**: ICP is essential for SLAM (Simultaneous Localization and Mapping), where robots need to build maps while tracking their position. It enables real-time registration of sensor data like LiDAR scans

### Robust Kernel

This tutorial demonstrates the use of robust kernels in the context of outlier rejection. For this particular tutorial, we will be using the ICP (Iterative Closest Point) registration algorithm as the *target* problem where we want to deal with outliers.

## Reconstruction system

### System overview

The system has 4 main steps:

1. **Make fragments**: build local geometric surfaces (referred to as fragments) from short subsequences of the input RGBD sequence. This part uses RGBD Odometry, Multiway registration, and RGBD integration.
2. **Register fragments**: the fragments are aligned in a global space to detect loop closure. This part uses Global registration, ICP registration, and Multiway registration.
3. **Refine registration**: the rough alignments are aligned more tightly. This part uses ICP registration, and Multiway registration.
4. **Integrate scene**: integrate RGB-D images to generate a mesh model for the scene. This part uses RGBD integration.

### Make fragments

The first step of the scene reconstruction system is to create fragments from short RGBD sequences

Claude: The "short RGBD sequences" refer to **temporal sequences** - meaning the robot (or camera) has moved and captured multiple RGB-D frames over time from different viewpoints of the same scene.

**RGBD data** (RGB color + Depth), which is common from:

* RGB-D cameras like Kinect, RealSense
* Processed stereo camera data
* Some mobile phone depth sensors

#### Register RGBD image pairs

def register\_one\_rgbd\_pair(s, t, color\_files, depth\_files, intrinsic,

with\_opencv, config):

source\_rgbd\_image = read\_rgbd\_image(color\_files[s], depth\_files[s], True,

config)

target\_rgbd\_image = read\_rgbd\_image(color\_files[t], depth\_files[t], True,

config)

option = o3d.pipelines.odometry.OdometryOption()

option.depth\_diff\_max = config["depth\_diff\_max"]

if abs(s - t) != 1:

if with\_opencv:

success\_5pt, odo\_init = pose\_estimation(source\_rgbd\_image,

target\_rgbd\_image,

intrinsic, False)

if success\_5pt:

[success, trans, info

] = o3d.pipelines.odometry.compute\_rgbd\_odometry(

source\_rgbd\_image, target\_rgbd\_image, intrinsic, odo\_init,

o3d.pipelines.odometry.RGBDOdometryJacobianFromHybridTerm(),

option)

return [success, trans, info]

return [False, np.identity(4), np.identity(6)]

else:

odo\_init = np.identity(4)

[success, trans, info] = o3d.pipelines.odometry.compute\_rgbd\_odometry(

source\_rgbd\_image, target\_rgbd\_image, intrinsic, odo\_init,

o3d.pipelines.odometry.RGBDOdometryJacobianFromHybridTerm(), option)

return [success, trans, info]

The function reads a pair of RGBD images and registers the source\_rgbd\_image to the target\_rgbd\_image. The Open3D function **compute\_rgbd\_odometry** is called to align the RGBD images. For adjacent RGBD images, an identity matrix is used as the initialization. For non-adjacent RGBD images, wide baseline matching is used as the initialization. In particular, the function **pose\_estimation** computes OpenCV ORB feature to match sparse features over wide baseline images, then performs 5-point RANSAC to estimate a rough alignment, which is used as the initialization of compute\_rgbd\_odometry.

#### Multiway registration

def **make\_posegraph\_for\_fragment**(path\_dataset, sid, eid, color\_files,

depth\_files, fragment\_id, n\_fragments,

intrinsic, with\_opencv, config):

o3d.utility.set\_verbosity\_level(o3d.utility.VerbosityLevel.Error)

pose\_graph = o3d.pipelines.registration.PoseGraph()

trans\_odometry = np.identity(4)

pose\_graph.nodes.append(

o3d.pipelines.registration.PoseGraphNode(trans\_odometry))

for s in range(sid, eid):

for t in range(s + 1, eid):

# odometry

if t == s + 1:

print(

"Fragment %03d / %03d :: RGBD matching between frame : %d and %d"

% (fragment\_id, n\_fragments - 1, s, t))

[success, trans,

info] = register\_one\_rgbd\_pair(s, t, color\_files, depth\_files,

intrinsic, with\_opencv, config)

trans\_odometry = np.dot(trans, trans\_odometry)

trans\_odometry\_inv = np.linalg.inv(trans\_odometry)

pose\_graph.nodes.append(

o3d.pipelines.registration.PoseGraphNode(

trans\_odometry\_inv))

pose\_graph.edges.append(

o3d.pipelines.registration.PoseGraphEdge(s - sid,

t - sid,

trans,

info,

uncertain=False))

# keyframe loop closure

if s % config['n\_keyframes\_per\_n\_frame'] == 0 \

and t % config['n\_keyframes\_per\_n\_frame'] == 0:

print(

"Fragment %03d / %03d :: RGBD matching between frame : %d and %d"

% (fragment\_id, n\_fragments - 1, s, t))

[success, trans,

info] = register\_one\_rgbd\_pair(s, t, color\_files, depth\_files,

intrinsic, with\_opencv, config)

if success:

pose\_graph.edges.append(

o3d.pipelines.registration.PoseGraphEdge(

s - sid, t - sid, trans, info, uncertain=True))

o3d.io.write\_pose\_graph(

join(path\_dataset, config["template\_fragment\_posegraph"] % fragment\_id),

pose\_graph)

This script uses the technique demonstrated in Multiway registration. The function make\_posegraph\_for\_fragment builds a pose graph for multiway registration of all RGBD images in this sequence. Each graph node represents an RGBD image and its pose which transforms the geometry to the global fragment space. For efficiency, only key frames are used.

Once a pose graph is created, multiway registration is performed by calling the function optimize\_posegraph\_for\_fragment shown below.

def **optimize\_posegraph\_for\_fragment**(path\_dataset, fragment\_id, config):

pose\_graph\_name = join(path\_dataset,

config["template\_fragment\_posegraph"] % fragment\_id)

pose\_graph\_optimized\_name = join(

path\_dataset,

config["template\_fragment\_posegraph\_optimized"] % fragment\_id)

run\_posegraph\_optimization(pose\_graph\_name, pose\_graph\_optimized\_name,

max\_correspondence\_distance = config["depth\_diff\_max"],

preference\_loop\_closure = \

config["preference\_loop\_closure\_odometry"])

#### Make a fragment

def integrate\_rgb\_frames\_for\_fragment(color\_files, depth\_files, fragment\_id,

n\_fragments, pose\_graph\_name, intrinsic,

config):

pose\_graph = o3d.io.read\_pose\_graph(pose\_graph\_name)

volume = o3d.pipelines.integration.ScalableTSDFVolume(

voxel\_length=config["tsdf\_cubic\_size"] / 512.0,

sdf\_trunc=0.04,

color\_type=o3d.pipelines.integration.TSDFVolumeColorType.RGB8)

for i in range(len(pose\_graph.nodes)):

i\_abs = fragment\_id \* config['n\_frames\_per\_fragment'] + i

print(

"Fragment %03d / %03d :: integrate rgbd frame %d (%d of %d)." %

(fragment\_id, n\_fragments - 1, i\_abs, i + 1, len(pose\_graph.nodes)))

rgbd = read\_rgbd\_image(color\_files[i\_abs], depth\_files[i\_abs], False,

config)

pose = pose\_graph.nodes[i].pose

volume.integrate(rgbd, intrinsic, np.linalg.inv(pose))

mesh = volume.extract\_triangle\_mesh()

mesh.compute\_vertex\_normals()

return mesh

Once the poses are estimated, [RGBD integration](https://www.open3d.org/docs/release/tutorial/pipelines/rgbd_integration.html) is used to reconstruct a colored fragment from each RGBD sequence.

### Register fragments

Once the fragments of the scene are created, the next step is to align them in a global space.

#### Input arguments

This script runs with python run\_system.py [config] --register. In [config], ["path\_dataset"] should have subfolders fragments which stores fragments in .ply files and a pose graph in a .json file.

The main function runs make\_posegraph\_for\_scene and optimize\_posegraph\_for\_scene. The first function performs pairwise registration. The second function performs multiway registration.

#### Preprocess point cloud

def preprocess\_point\_cloud(pcd, config):

voxel\_size = config["voxel\_size"]

pcd\_down = pcd.voxel\_down\_sample(voxel\_size)

pcd\_down.estimate\_normals(

o3d.geometry.KDTreeSearchParamHybrid(radius=voxel\_size \* 2.0,

max\_nn=30))

pcd\_fpfh = o3d.pipelines.registration.compute\_fpfh\_feature(

pcd\_down,

o3d.geometry.KDTreeSearchParamHybrid(radius=voxel\_size \* 5.0,

max\_nn=100))

return (pcd\_down, pcd\_fpfh)

This function downsamples a point cloud to make it sparser and regularly distributed.

#### Compute initial registration

def compute\_initial\_registration(s, t, source\_down, target\_down, source\_fpfh,

target\_fpfh, path\_dataset, config):

if t == s + 1: # odometry case

print("Using RGBD odometry")

pose\_graph\_frag = o3d.io.read\_pose\_graph(

join(path\_dataset,

config["template\_fragment\_posegraph\_optimized"] % s))

n\_nodes = len(pose\_graph\_frag.nodes)

transformation\_init = np.linalg.inv(pose\_graph\_frag.nodes[n\_nodes -

1].pose)

(transformation, information) = \

multiscale\_icp(source\_down, target\_down,

[config["voxel\_size"]], [50], config, transformation\_init)

else: # loop closure case

(success, transformation,

information) = register\_point\_cloud\_fpfh(source\_down, target\_down,

source\_fpfh, target\_fpfh,

config)

if not success:

print("No reasonable solution. Skip this pair")

return (False, np.identity(4), np.zeros((6, 6)))

print(transformation)

if config["debug\_mode"]:

draw\_registration\_result(source\_down, target\_down, transformation)

return (True, transformation, information)

This function computes a rough alignment between two fragments. If the fragments are neighboring fragments, the rough alignment is determined by an aggregating RGBD odometry obtained from Make fragments. Otherwise, **register\_point\_cloud\_fpfh** is called to perform global registration.

#### Pairwise global registration

def register\_point\_cloud\_fpfh(source, target, source\_fpfh, target\_fpfh, config):

o3d.utility.set\_verbosity\_level(o3d.utility.VerbosityLevel.Debug)

distance\_threshold = config["voxel\_size"] \* 1.4

if config["global\_registration"] == "fgr":

result = o3d.pipelines.registration.registration\_fgr\_based\_on\_feature\_matching(

source, target, source\_fpfh, target\_fpfh,

o3d.pipelines.registration.FastGlobalRegistrationOption(

maximum\_correspondence\_distance=distance\_threshold))

if config["global\_registration"] == "ransac":

# Fallback to preset parameters that works better

result = o3d.pipelines.registration.registration\_ransac\_based\_on\_feature\_matching(

source, target, source\_fpfh, target\_fpfh, False, distance\_threshold,

o3d.pipelines.registration.TransformationEstimationPointToPoint(

False), 4,

[

o3d.pipelines.registration.

CorrespondenceCheckerBasedOnEdgeLength(0.9),

o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(

distance\_threshold)

],

o3d.pipelines.registration.RANSACConvergenceCriteria(

1000000, 0.999))

if (result.transformation.trace() == 4.0):

return (False, np.identity(4), np.zeros((6, 6)))

information = o3d.pipelines.registration.get\_information\_matrix\_from\_point\_clouds(

source, target, distance\_threshold, result.transformation)

if information[5, 5] / min(len(source.points), len(target.points)) < 0.3:

return (False, np.identity(4), np.zeros((6, 6)))

return (True, result.transformation, information)

This function uses **RANSAC** or **Fast global registration** for pairwise global registration.

#### Multiway registration

def **update\_posegraph\_for\_scene**(s, t, transformation, information, odometry,

pose\_graph):

if t == s + 1: # odometry case

odometry = np.dot(transformation, odometry)

odometry\_inv = np.linalg.inv(odometry)

pose\_graph.nodes.append(

o3d.pipelines.registration.PoseGraphNode(odometry\_inv))

pose\_graph.edges.append(

o3d.pipelines.registration.PoseGraphEdge(s,

t,

transformation,

information,

uncertain=False))

else: # loop closure case

pose\_graph.edges.append(

o3d.pipelines.registration.PoseGraphEdge(s,

t,

transformation,

information,

uncertain=True))

return (odometry, pose\_graph)

This script uses the technique demonstrated in Multiway registration. The function update\_posegraph\_for\_scene (above) builds a pose graph for multiway registration of all fragments. Each graph node represents a fragment and its pose which transforms the geometry to the global space.

Once a pose graph is built, the function optimize\_posegraph\_for\_scene (below) is called for multiway registration.

def optimize\_posegraph\_for\_scene(path\_dataset, config):

pose\_graph\_name = join(path\_dataset, config["template\_global\_posegraph"])

pose\_graph\_optimized\_name = join(

path\_dataset, config["template\_global\_posegraph\_optimized"])

run\_posegraph\_optimization(pose\_graph\_name, pose\_graph\_optimized\_name,

max\_correspondence\_distance = config["voxel\_size"] \* 1.4,

preference\_loop\_closure = \

config["preference\_loop\_closure\_registration"])

#### Main registration loop

The function make\_posegraph\_for\_scene below calls all the functions introduced above. The main workflow is: *pairwise global registration -> multiway registration*.

def **make\_posegraph\_for\_scene**(ply\_file\_names, config):

pose\_graph = o3d.pipelines.registration.PoseGraph()

odometry = np.identity(4)

pose\_graph.nodes.append(o3d.pipelines.registration.PoseGraphNode(odometry))

n\_files = len(ply\_file\_names)

matching\_results = {}

for s in range(n\_files):

for t in range(s + 1, n\_files):

matching\_results[s \* n\_files + t] = matching\_result(s, t)

if config["python\_multi\_threading"] is True:

os.environ['OMP\_NUM\_THREADS'] = '1'

max\_workers = max(

1, min(multiprocessing.cpu\_count() - 1, len(matching\_results)))

mp\_context = multiprocessing.get\_context('spawn')

with mp\_context.Pool(processes=max\_workers) as pool:

args = [(ply\_file\_names, v.s, v.t, config)

for k, v in matching\_results.items()]

results = pool.starmap(register\_point\_cloud\_pair, args)

for i, r in enumerate(matching\_results):

matching\_results[r].success = results[i][0]

matching\_results[r].transformation = results[i][1]

matching\_results[r].information = results[i][2]

else:

for r in matching\_results:

(matching\_results[r].success, matching\_results[r].transformation,

matching\_results[r].information) = \

register\_point\_cloud\_pair(ply\_file\_names,

matching\_results[r].s, matching\_results[r].t, config)

for r in matching\_results:

if matching\_results[r].success:

(odometry, pose\_graph) = update\_posegraph\_for\_scene(

matching\_results[r].s, matching\_results[r].t,

matching\_results[r].transformation,

matching\_results[r].information, odometry, pose\_graph)

o3d.io.write\_pose\_graph(

join(config["path\_dataset"], config["template\_global\_posegraph"]),

pose\_graph)

### Refine registration

#### Input arguments

This script runs with python run\_system.py [config] --refine. In [config], ["path\_dataset"] should have subfolders fragments which stores fragments in .ply files and a pose graph in a .json file.

The main function runs local\_refinement and optimize\_posegraph\_for\_scene. The first function performs pairwise registration on the pairs detected by **Register fragments**. The second function performs multiway registration.

#### Fine-grained registration

#### Multiway registration

#### Main registration loop

### Integrate scene

The final step of the system is to integrate all RGBD images into a single TSDF volume and extract a mesh as the result.

#### Input arguments

#### Integrate RGBD frames

def scalable\_integrate\_rgb\_frames(path\_dataset, intrinsic, config):

poses = []

[color\_files, depth\_files] = get\_rgbd\_file\_lists(path\_dataset)

n\_files = len(color\_files)

n\_fragments = int(math.ceil(float(n\_files) / \

config['n\_frames\_per\_fragment']))

volume = o3d.pipelines.integration.ScalableTSDFVolume(

voxel\_length=config["tsdf\_cubic\_size"] / 512.0,

sdf\_trunc=0.04,

color\_type=o3d.pipelines.integration.TSDFVolumeColorType.RGB8)

pose\_graph\_fragment = o3d.io.read\_pose\_graph(

join(path\_dataset, config["template\_refined\_posegraph\_optimized"]))

for fragment\_id in range(len(pose\_graph\_fragment.nodes)):

pose\_graph\_rgbd = o3d.io.read\_pose\_graph(

join(path\_dataset,

config["template\_fragment\_posegraph\_optimized"] % fragment\_id))

for frame\_id in range(len(pose\_graph\_rgbd.nodes)):

frame\_id\_abs = fragment\_id \* \

config['n\_frames\_per\_fragment'] + frame\_id

print(

"Fragment %03d / %03d :: integrate rgbd frame %d (%d of %d)." %

(fragment\_id, n\_fragments - 1, frame\_id\_abs, frame\_id + 1,

len(pose\_graph\_rgbd.nodes)))

rgbd = read\_rgbd\_image(color\_files[frame\_id\_abs],

depth\_files[frame\_id\_abs], False, config)

pose = np.dot(pose\_graph\_fragment.nodes[fragment\_id].pose,

pose\_graph\_rgbd.nodes[frame\_id].pose)

volume.integrate(rgbd, intrinsic, np.linalg.inv(pose))

poses.append(pose)

mesh = volume.extract\_triangle\_mesh()

mesh.compute\_vertex\_normals()

if config["debug\_mode"]:

o3d.visualization.draw\_geometries([mesh])

This function first reads the alignment results from both Make fragments and Register fragments, then computes the pose of each RGBD image in the global space. After that, RGBD images are integrated using RGBD integration.

### Capture your own dataset

If you have a RealSense camera, capturing RGBD frames is easy by using sensors/realsense\_recorder.py

#### Input arguments

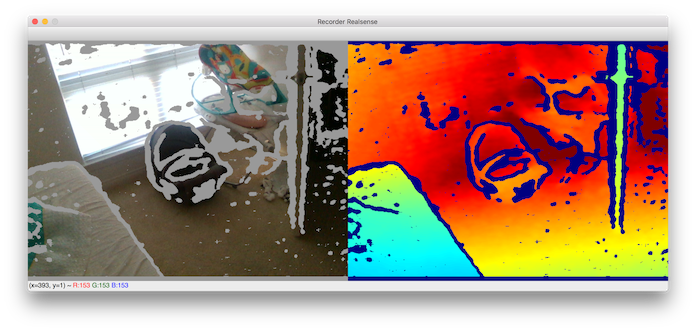
The script runs with one of the following three options:

python realsense\_recorder.py --record\_imgs

python realsense\_recorder.py --record\_rosbag

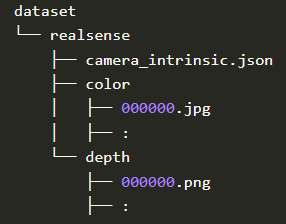
python realsense\_recorder.py --playback\_rosbag

In either record\_imgs and record\_rosbag mode, the script displays the following capturing preview.



The left side shows color image with invalid depth region markup (in gray color), and the right side shows jet color coded depth map. Invalid depth pixels are object boundary, uncertain region, or distant region (more than 3m). Capturing frames without too many gray pixels is recommended for the good reconstruction quality.

By default, record\_imgs mode saves aligned color and depth images in dataset/realsense folder that can be used for reconstruction system.



#### Make a new configuration file

A new configuration file is required to specify path to the new dataset. config/realsense.json is provided for this purpose.

{

"name": "Realsense bag file",

"path\_dataset": "dataset/realsense.bag",

"path\_intrinsic": "optional/read\_from\_bag\_file.json",

"depth\_max": 3.0,

"voxel\_size": 0.05,

"depth\_diff\_max": 0.07,

"preference\_loop\_closure\_odometry": 0.1,

"preference\_loop\_closure\_registration": 5.0,

"tsdf\_cubic\_size": 3.0,

"icp\_method": "color",

"global\_registration": "ransac",

"python\_multi\_threading": true

}

Note that path\_dataset and path\_intrinsic indicates paths of dataset and intrinsic parameters.

#### Run reconstruction system

Run the system by using the new configuration file.

cd examples/python/reconstruction\_system/

python run\_system.py config/realsense.json [--make] [--register] [--refine] [--integrate]

## Reconstruction system (Tensor)

### Voxel Block Grid

A voxel block grid is a globally sparse and locally dense data structure to represent 3D scenes. It is globally sparse since 2D object surfaces are usually occupying a small portion of the 3D space; it is locally dense in order to represent contiguous surfaces.

To represent such a structure, we first coarsely divide the 3D space into **block** grids. Blocks containing surfaces are organized in a hash map by 3D coordinates (sparse globally), and are further divided into dense **voxels** that can be accessed by array indices (dense locally

Construction

A voxel block grid can be constructed by:

# examples/python/t\_reconstruction\_system/integrate.py

if config.integrate\_color:

vbg = o3d.t.geometry.VoxelBlockGrid(

attr\_names=('tsdf', 'weight', 'color'),

attr\_dtypes=(o3c.float32, o3c.float32, o3c.float32),

attr\_channels=((1), (1), (3)),

voxel\_size=3.0 / 512,

block\_resolution=16,

block\_count=50000,

device=device)

else:

vbg = o3d.t.geometry.VoxelBlockGrid(attr\_names=('tsdf', 'weight'),

attr\_dtypes=(o3c.float32,

o3c.float32),

attr\_channels=((1), (1)),

voxel\_size=3.0 / 512,

block\_resolution=16,

block\_count=50000,

device=device)

### TSDF Integration

***Truncated Signed Distance Function (TSDF)*** integration is the key of dense volumetric scene reconstruction. It receives relatively noisy depth images from RGB-D sensors such as Kinect and RealSense, and integrates depth readings into the **Voxel Block Grid** given known camera poses. TSDF integration reduces noise and generates smooth surfaces.

The integration process mainly consists of two steps, (sparse) ***block****selection and activation*, and (dense*)****voxel****value integration*. An example can be found at examples/python/t\_reconstruction\_system/integrate.py.

#### Claude Q: What is Truncated Signed Distance Function (TSDF)?

**Truncated Signed Distance Function (TSDF)** is a 3D representation technique that's become fundamental in computer vision and robotics for reconstructing and understanding 3D environments from sensor data.

Here's what it is: A signed distance function (SDF) stores, for every point in 3D space, the shortest distance to the nearest surface. The "signed" part means the distance is positive outside objects and negative inside them. The "truncated" part means you only store distance values within a certain range around surfaces (typically a few centimeters), setting everything else to the truncation limit.

Think of it like this: imagine you're mapping a room with a depth camera. Instead of just storing individual 3D points, TSDF creates a 3D grid where each cell contains how far that location is from the nearest wall or object surface. Near surfaces, you get precise distance measurements. Far from surfaces, you just store "more than X distance away."

Why you should care:

**Real-time 3D reconstruction** - TSDF enables systems like Microsoft's Kinect Fusion to build detailed 3D models of rooms as you move a camera around. The incremental fusion of new depth measurements into the TSDF volume creates increasingly accurate reconstructions.

**Robotics and SLAM** - Robots use TSDF for simultaneous localization and mapping. It helps them understand their environment's geometry for navigation and manipulation tasks.

#### Activation

In the activation step, we first locate blocks that contain points unprojected from the current depth image. In other words, it finds active blocks in the current viewing frustum. Internally, this is achieved by a *frustum* hash map that produces duplication-free block coordinates, and a *block* hash map that activates and query such block coordinates.

# examples/python/t\_reconstruction\_system/integrate.py

frustum\_block\_coords = vbg.compute\_unique\_block\_coordinates(

depth, depth\_intrinsic, extrinsic, config.depth\_scale,

config.depth\_max)

#### Integration

Now we can process the voxels in the blocks at frustum\_block\_coords. This is done by projecting all such related voxels to the input images and perform a weighted average, which is a pure geometric process without hash map operations.

We may use optimized functions, along with raw depth images with calibration parameters to activate and perform TSDF integration, optionally with colors:

# examples/python/t\_reconstruction\_system/integrate.py

if config.integrate\_color:

color = o3d.t.io.read\_image(color\_file\_names[i]).to(device)

vbg.integrate(frustum\_block\_coords, depth, color, depth\_intrinsic,

color\_intrinsic, extrinsic, config.depth\_scale,

config.depth\_max)

else:

vbg.integrate(frustum\_block\_coords, depth, depth\_intrinsic,

extrinsic, config.depth\_scale, config.depth\_max)

#### Surface extraction

You may use the provided APIs to extract surface points.

# examples/python/t\_reconstruction\_system/integrate.py

pcd = vbg.extract\_point\_cloud()

o3d.visualization.draw([pcd])

mesh = vbg.extract\_triangle\_mesh()

o3d.visualization.draw([mesh.to\_legacy()])

#### Save and load

The voxel block grids can be saved to and loaded from *.npz* files that are accessible via numpy.

### Customized Integration

You can prototype a new RGB-D volumetric reconstruction algorithm with additional properties (e.g. semantic labels) while maintaining a reasonable performance. An example can be found at examples/python/t\_reconstruction\_system/integrate\_custom.py.

### Ray Casting in a Voxel Block Grid

Ray casting can be performed in a voxel block grid to generate depth and color images at specific view points without extracting the entire surface. It is useful for frame-to-model tracking, and for differentiable volume rendering.

We provide optimized conventional rendering, and basic support for customized rendering that may be used in differentiable rendering. An example can be found at examples/python/t\_reconstruction\_system/ray\_casting.py.

### Dense RGB-D SLAM

Equipped with the fast volumetric reconstruction backend, we in addition provide a dense RGB-D SLAM system using frame-to-model tracking. The example can be found at examples/python/t\_reconstruction\_system/dense\_slam.py for the command line version and examples/python/t\_reconstruction\_system/dense\_slam\_gui.py for a GUI demo.

## Sensor

### Azure Kinect with Open3D

Azure Kinect is only officially supported on Windows and Ubuntu 18.04.

**Install Open3D from Pip**

If you’re using Open3D installed via Pip, Open3D’s Azure Kinect features should work out-of-the box if K4A is installed in the system in the recommended way. Open3D will try to load the K4A dynamic library automatically at runtime, when a K4A related feature within Open3D is used.

On Ubuntu, the default search path follows the Linux convention.

On Windows, Open3D will try to load the shared library from the default installation path. For example, for K4A v1.4.1, the default path is C:\Program Files\Azure Kinect SDK v1.4.1. If this doesn’t work, copy depthengine\_x\_x.dll, k4a.dll and k4arecord.dll to where Open3D Python module is installed if you’re using Open3D with Python, or to the same directory as your C++ executable.

**Compile from Source**

To build Open3D from source with K4A support, set BUILD\_AZURE\_KINECT=ON at CMake config step. That is,

cmake -DBUILD\_AZURE\_KINECT=ON -DOTHER\_FLAGS ..

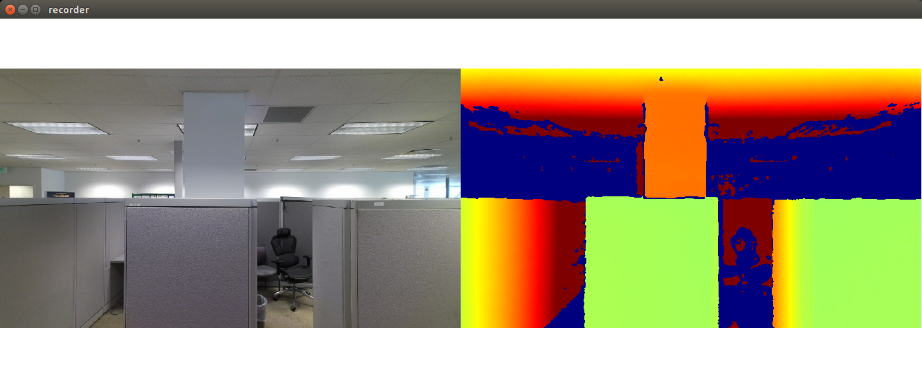
**Open3D Azure Kinect Viewer**

Open3D Azure Kinect Viewer is used for previewing RGB and depth image stream captured by the Azure Kinect sensor.

Open3D provides Python and C++ example code of Azure Kinect viewer. Please see examples/cpp/AzureKinectViewer.cpp and examples/python/reconstruction\_system/sensors/azure\_kinect\_viewer.py for details.

We’ll use the Python version as an example.

python examples/python/reconstruction\_system/sensors/azure\_kinect\_viewer.py --align\_depth\_to\_color



An sensor config will look like the following.

{

"camera\_fps" : "K4A\_FRAMES\_PER\_SECOND\_30",

"color\_format" : "K4A\_IMAGE\_FORMAT\_COLOR\_MJPG",

"color\_resolution" : "K4A\_COLOR\_RESOLUTION\_720P",

"depth\_delay\_off\_color\_usec" : "0",

"depth\_mode" : "K4A\_DEPTH\_MODE\_WFOV\_2X2BINNED",

"disable\_streaming\_indicator" : "false",

"subordinate\_delay\_off\_master\_usec" : "0",

"synchronized\_images\_only" : "false",

"wired\_sync\_mode" : "K4A\_WIRED\_SYNC\_MODE\_STANDALONE"

}

### RealSense with Open3D

RealSense (librealsense SDK v2) is integrated into Open3D (v0.12+) and you can use it through both C++ and Python APIs without a separate librealsense SDK installation

**Reading from RealSense bag files**

**Sample RealSense bag files**

You con download sample RealSense bag datasets with this script:

python examples/python/reconstruction\_system/scripts/download\_dataset.py L515\_test

#include <open3d/open3d.hpp>

using namespace open3d;

t::io::RSBagReader bag\_reader;

bag\_reader.Open(bag\_filename);

auto im\_rgbd = bag\_reader.NextFrame();

while (!bag\_reader.IsEOF()) {

// process im\_rgbd.depth\_ and im\_rgbd.color\_

im\_rgbd = bag\_reader.NextFrame();

}

bag\_reader.Close();

**Running the scene reconstruction pipeline**

You can provide an RS bag file directly to the reconstruction pipeline and colormap optimization pipelines. It will be automatically converted to a directory of depth and color frames and the camera intrinsics.

cd examples/python/reconstruction\_system/

python run\_system.py --make --register --refine --integrate config/realsense.json

python color\_map\_optimization\_for\_reconstruction\_system.py --config config/realsense.json

#### RealSense camera configuration, live capture, processing and recording

**RealSense camera discovery**

You can list all RealSense devices connected to the system and get their capabilities (supported resolutions, frame rates, etc.) with the code snippet below.

#include <open3d/open3d.hpp>

open3d::t::io::RealSenseSensor::ListDevices();

Python:

import open3d as o3d

o3d.t.io.RealSenseSensor.list\_devices()

**RealSense camera configuration**

RealSense cameras can be configured with a simple json configuration file.

Here are the options supported by Open3D:

* **serial**: Pick a specific device, leave empty to pick the first available device.
* **color\_format**: Pixel format for color frames.
* **color\_resolution**: (width, height): Leave 0 to let RealSense pick a supported width or height.
* **depth\_format**: Pixel format for depth frames.
* **depth\_resolution**: (width, height): Leave 0 to let RealSense pick a supported width or height.
* **fps**: Common frame rate for both depth and color streams. Leave 0 to let RealSense pick a supported frame rate.
* **visual\_preset**: Controls depth computation on the device. Supported values are specific to product line (SR300, RS400, L500). Leave empty to pick the default.

Here is an example json configuration file

{

"serial": "",

"color\_format": "RS2\_FORMAT\_RGB8",

"color\_resolution": "0,540",

"depth\_format": "RS2\_FORMAT\_Z16",

"depth\_resolution": "0,480",

"fps": "30",

"visual\_preset": "RS2\_L500\_VISUAL\_PRESET\_MAX\_RANGE"

}

**RealSense camera capture, processing and recording**

The following code snippets show how to capture live RGBD video from a RealSense camera. They capture the first 150 frames and also record them to an RS bag file. The bag file can be played back with Open3D tools, realsense-viewer. You can also use ROS tools such as [rosbag](http://wiki.ros.org/rosbag), [rqt\_bag](http://wiki.ros.org/rqt_bag) and [rviz](https://wiki.ros.org/rviz) to examine, play and modify the bag file.

import json

import open3d as o3d

with open(config\_filename) as cf:

rs\_cfg = o3d.t.io.RealSenseSensorConfig(json.load(cf))

rs = o3d.t.io.RealSenseSensor()

rs.init\_sensor(rs\_cfg, 0, bag\_filename)

rs.start\_capture(True) # true: start recording with capture

for fid in range(150):

im\_rgbd = rs.capture\_frame(True, True) # wait for frames and align them

# process im\_rgbd.depth and im\_rgbd.color

rs.stop\_capture()

## Reference

# Python API

# Python Examples

# Open3D-ML

<https://github.com/isl-org/Open3D-ML>

## Semantic Segmentation

Running a pretrained model for semantic segmentation

import os

import open3d.ml as \_ml3d

import open3d.ml.torch as ml3d

cfg\_file = "ml3d/configs/randlanet\_semantickitti.yml"

cfg = \_ml3d.utils.Config.load\_from\_file(cfg\_file)

model = ml3d.models.RandLANet(\*\*cfg.model)

cfg.dataset['dataset\_path'] = "/path/to/your/dataset"

dataset = ml3d.datasets.SemanticKITTI(cfg.dataset.pop('dataset\_path', None), \*\*cfg.dataset)

pipeline = ml3d.pipelines.SemanticSegmentation(model, dataset=dataset, device="gpu", \*\*cfg.pipeline)

# download the weights.

ckpt\_folder = "./logs/"

os.makedirs(ckpt\_folder, exist\_ok=True)

ckpt\_path = ckpt\_folder + "randlanet\_semantickitti\_202201071330utc.pth"

randlanet\_url =

"https://storage.googleapis.com/open3d-releases/model-zoo/randlanet\_semantickitti\_202201071330utc.pth"

if not os.path.exists(ckpt\_path):

cmd = "wget {} -O {}".format(randlanet\_url, ckpt\_path)

os.system(cmd)

# load the parameters.

pipeline.load\_ckpt(ckpt\_path=ckpt\_path)

test\_split = dataset.get\_split("test")

data = test\_split.get\_data(0)

# run inference on a single example.

# returns dict with 'predict\_labels' and 'predict\_scores'.

result = pipeline.run\_inference(data)

# evaluate performance on the test set; this will write logs to './logs'.

pipeline.run\_test()

## 3D Object Detection

Running a pretrained model for 3D object detection

The 3D object detection model is similar to a semantic segmentation model. We can instantiate a pipeline with a pretrained model for Object Detection and run it on a point cloud of our dataset. See the [model zoo](https://github.com/isl-org/Open3D-ML#model-zoo) for obtaining the weights of the pretrained model.

import os

import open3d.ml as \_ml3d

import open3d.ml.torch as ml3d

cfg\_file = "ml3d/configs/pointpillars\_kitti.yml"

cfg = \_ml3d.utils.Config.load\_from\_file(cfg\_file)

model = ml3d.models.PointPillars(\*\*cfg.model)

cfg.dataset['dataset\_path'] = "/path/to/your/dataset"

dataset = ml3d.datasets.KITTI(cfg.dataset.pop('dataset\_path', None), \*\*cfg.dataset)

pipeline = ml3d.pipelines.ObjectDetection(model, dataset=dataset, device="gpu", \*\*cfg.pipeline)

# download the weights.

ckpt\_folder = "./logs/"

os.makedirs(ckpt\_folder, exist\_ok=True)

ckpt\_path = ckpt\_folder + "pointpillars\_kitti\_202012221652utc.pth"

pointpillar\_url =

"https://storage.googleapis.com/open3d-releases/model-zoo/pointpillars\_kitti\_202012221652utc.pth"

if not os.path.exists(ckpt\_path):

cmd = "wget {} -O {}".format(pointpillar\_url, ckpt\_path)

os.system(cmd)

# load the parameters.

pipeline.load\_ckpt(ckpt\_path=ckpt\_path)

test\_split = dataset.get\_split("test")

data = test\_split.get\_data(0)

# run inference on a single example.

# returns dict with 'predict\_labels' and 'predict\_scores'.

result = pipeline.run\_inference(data)

# evaluate performance on the test set; this will write logs to './logs'.

pipeline.run\_test()

### Explanation from Claude

#### Overview

This Open3D code demonstrates a complete 3D object detection pipeline using the PointPillars architecture on the KITTI dataset. Let me walk you through each component to help you understand how 3D object detection works in practice.

Understanding 3D Object Detection

Before diving into the code, it's helpful to understand what makes 3D object detection different from traditional 2D image detection. In 3D object detection, we're working with point clouds - collections of 3D points in space that represent the geometry of a scene. These point clouds are typically captured by LiDAR sensors, which emit laser pulses and measure the time it takes for them to return, creating precise 3D measurements of the environment.

The challenge is that point clouds are unstructured data - unlike images which have a regular grid of pixels, point clouds are sparse and irregular. This is where PointPillars comes in as an elegant solution.

The PointPillars Architecture

PointPillars transforms the irregular 3D point cloud into a structured representation that can be processed by conventional 2D convolutional neural networks. Think of it as creating "pillars" or vertical columns in 3D space, where each pillar contains statistical summaries of all the points within that column. This approach bridges the gap between 3D data and 2D processing techniques.

Code Breakdown

Let's examine each section of your code:

Configuration and Model Setup:

cfg\_file = "ml3d/configs/pointpillars\_kitti.yml"

cfg = ml3d.utils.Config.load\_from\_file(cfg\_file)

model = ml3d.models.PointPillars(\*\*cfg.model)

This section loads a configuration file that contains all the hyperparameters and settings for the PointPillars model. Configuration files are crucial in machine learning because they allow you to experiment with different settings without modifying the core code. The KITTI configuration is specifically tuned for autonomous driving scenarios.

Dataset Preparation:

cfg.dataset['dataset\_path'] = "/path/to/your/dataset"

dataset = ml3d.datasets.KITTI(cfg.dataset.pop('dataset\_path', None), \*\*cfg.dataset)

The KITTI dataset is one of the most important benchmarks in autonomous driving research. It contains point cloud data from urban environments with ground truth annotations for cars, pedestrians, and cyclists. The dataset class handles all the complex preprocessing needed to convert raw point clouds into the format expected by PointPillars.

Pipeline Creation:

pipeline = ml3d.pipelines.ObjectDetection(model, dataset=dataset,

device="gpu", \*\*cfg.pipeline)

The pipeline is where everything comes together. It combines the model, dataset, and processing logic into a unified interface. Think of it as the conductor of an orchestra - it coordinates all the different components to work harmoniously together.

Model Weight Loading:

# download the weights and load them

ckpt\_path = ckpt\_folder + "pointpillars\_kitti\_202012221652utc.pth"

pipeline.load\_ckpt(ckpt\_path=ckpt\_path)

This section downloads pre-trained weights. Training a 3D object detection model from scratch requires enormous computational resources and time, so using pre-trained weights is essential for practical applications. These weights represent millions of learned parameters that encode the knowledge of how to detect objects in 3D space.

Inference and Evaluation:

result = pipeline.run\_inference(data)

pipeline.run\_test()

The inference step is where the magic happens - the model processes a single point cloud and returns predictions about where objects are located in 3D space, what type of objects they are, and how confident the model is about each prediction. The test evaluation runs the model on the entire test set to measure its performance using standard metrics.

The Bigger Picture

This code represents a complete workflow that you might find in a real autonomous vehicle system. The point cloud data would come from roof-mounted LiDAR sensors, and the detection results would feed into higher-level systems for path planning and decision making.

What makes this particularly interesting is how PointPillars solves the fundamental challenge of processing irregular 3D data efficiently. By creating a bird's-eye view representation through the pillar structure, it maintains spatial relationships while making the data suitable for fast 2D convolutions.

#### More details about "pipelines.ObjectDetection"?

Let me walk you through the pipelines.ObjectDetection class in detail, as it's really the heart of how Open3D-ML orchestrates the entire 3D object detection process.

Understanding the Pipeline Concept

Think of a pipeline as a factory assembly line for 3D object detection. Just as an assembly line has different stations that each perform specific tasks in a coordinated sequence, the ObjectDetection pipeline coordinates multiple complex operations that need to happen in the right order with the right data flow between them.

In traditional machine learning workflows, you often have to manually handle data loading, preprocessing, model inference, postprocessing, and evaluation. This becomes quite complex and error-prone, especially with 3D data where you're dealing with coordinate transformations, data augmentation, and multiple output formats. The pipeline abstracts away this complexity and provides a clean, unified interface.

Core Components and Responsibilities

When you create an ObjectDetection pipeline, you're essentially creating a coordinator that manages several key responsibilities. Let me break these down for you:

* **Data Flow Management**: The pipeline understands how to take raw point cloud data from your dataset and shepherd it through the entire detection process. This includes knowing what preprocessing steps need to happen, in what order, and how to batch multiple point clouds together efficiently for GPU processing.
* **Model Integration**: While your PointPillars model knows how to process tensors and produce outputs, it doesn't know about the broader context of evaluation metrics, coordinate systems, or how to interpret its own outputs in real-world terms. The pipeline bridges this gap by providing the model with properly formatted inputs and interpreting the model's outputs in meaningful ways.
* **Coordinate System Handling**: This is actually more complex than it might initially appear. Point clouds can exist in different coordinate systems depending on how they were captured and stored. The pipeline ensures that everything is aligned properly, handling transformations between sensor coordinates, world coordinates, and the normalized coordinate systems that the neural network expects.

#### The Training and Inference Modes (still from Claude)

The ObjectDetection pipeline operates in different modes depending on what you're trying to accomplish, and understanding these modes helps clarify its design.

**Training Mode**: When you're training a model, the pipeline coordinates the forward pass through the network, calculates losses by comparing predictions to ground truth labels, performs backpropagation, and updates model weights. It also handles complex data augmentation strategies like random rotations and scaling that are crucial for robust 3D object detection.

**Inference Mode**: This is what you're using in your code example. Here, the pipeline takes a point cloud, runs it through the trained model, and returns human-interpretable results. But there's more happening under the hood than just a simple forward pass.

Let me walk you through what happens when you call pipeline.run\_inference(data):

First, the pipeline takes your raw point cloud data and applies the same preprocessing that was used during training. This might include removing points that are too far away, normalizing coordinates, or converting the point cloud into the pillar representation that PointPillars expects.

Next, it runs the preprocessed data through your PointPillars model. The model outputs raw predictions in the form of classification scores, bounding box parameters, and confidence values. However, these outputs are in a format that's optimized for efficient neural network computation, not for human interpretation.

The pipeline then applies postprocessing to convert these raw outputs into meaningful object detections. This includes non-maximum suppression to remove duplicate detections, coordinate transformations to convert from the model's internal coordinate system back to real-world coordinates, and filtering based on confidence thresholds.

Configuration and Flexibility

One of the most powerful aspects of the ObjectDetection pipeline is how it uses configuration files to control its behavior. When you loaded that YAML configuration file, you weren't just setting model parameters - you were also configuring how the entire pipeline should operate.

The pipeline configuration might specify things like what data augmentation strategies to use during training, what coordinate transformations to apply, how to handle class imbalances in your dataset, and what evaluation metrics to compute. This design allows you to experiment with different pipeline configurations without modifying any code.

Evaluation and Metrics

When you call pipeline.run\_test(), you're activating the pipeline's evaluation capabilities. The pipeline will iterate through your entire test dataset, run inference on each example, and compute standard object detection metrics like Average Precision (AP) at different Intersection over Union (IoU) thresholds.

What makes 3D object detection evaluation particularly interesting is that it considers both the position and orientation of predicted bounding boxes in three-dimensional space. The pipeline handles all the geometric calculations needed to determine whether a predicted box sufficiently overlaps with a ground truth box to be considered a correct detection.

Device Management and Optimization

Notice how you specified device="gpu" when creating the pipeline. The pipeline automatically handles moving data between CPU and GPU memory, batching operations for efficient GPU utilization, and managing memory usage to prevent out-of-memory errors. This is particularly important for point cloud processing, where datasets can be quite large.

Think About This

Here's a mental exercise that might help solidify your understanding: imagine you had to implement this object detection workflow without the pipeline abstraction. You'd need to write code to load and preprocess point clouds, manage batching, handle coordinate transformations, implement postprocessing algorithms, compute evaluation metrics, and coordinate all of these components. The pipeline encapsulates all of this complexity into a clean interface.

The pipeline design also makes it easy to experiment with different models. If you wanted to try a different 3D object detection architecture instead of PointPillars, you could potentially just swap out the model while keeping the same pipeline, assuming the new model follows the same input/output conventions.